1985

An intermediate code for the translation of PASCAL and BCPL

Eliezer A. Albacea
University of Wollongong

An Intermediate Code for the Translation of PASCAL and BCPL

A thesis submitted in (partial) fulfilment of the

requirements for the award of the degree

Master of Science (Honours) in Computing Science

from

The University of Wollongong

by

Eliezer A. Albacea, BSc.

Department of Computing Science

1985

Abstract

This project responded to the question "Can PASCAL and BCPL be translated to the same intermediate code?" by translating PASCAL to an intermediate code which has been used only for the translation of BCPL.

The intermediate code that was used to achieve the translation is SLIM. Since SLIM was designed for BCPL, naturally, features supported in PASCAL but not in BCPL will certainly create difficulties in using SLIM as a target language for PASCAL. Several extensions to SLIM to make it a suitable target language for PASCAL and PASCAL-like languages are described.

Translation and optimization are two stages of the compilation process that compiler writers enjoy thinking about. The objective of most is to generate a reasonably efficient object code and at the same time to keep to a minimum the overhead introduced by the code generator during compilation.

The translator constructed to achieve the translation from PASCAL to SLIM is a strict one-pass translator. The translation presented in this thesis therefore can be generated in one pass. Alternative translations for a particular PASCAL source code fragment are presented. These alternative translations are designed to suit, as much as possible, with the nature of the machine where the program is to run.

Translating a high-level language is impossible without keeping extra information about the identifiers that were used by the source program. Hence, a thorough discussion of the creation and usage of

information from a symbol table is given.

Since a one-pass translation produces sub-optimal object code, the translator employs a "code improver" to obtain a more efficient object code. The code improver uses a peephole optimization technique. Three implementations of a peephole optimizer are discussed.

Finally, the conclusion discusses the suitability of SLIM as a target language for PASCAL and PASCAL-like languages.

TABLE OF CONTENTS

Acknowledgement

I would like to express my gratitude to J. E. L. Peck for his unending support during the development of this project, for editing this thesis, and for allowing me to use some of the tables in his book, "The Essence of Portable Programming".

My thanks to N. A. B. Gray for reading this thesis and for pointing out some of the things that I missed, to the Department of Computing Science for providing me the proper facilities for doing this research and to the students of Computing Science 333 (Compilers) Class '84 for helping me locate some of the "bugs" in the compiler.

Lastly, I am indebted to the International Development Program of Australian Universities and Colleges (IDP) and to the University of the Philippines at Los Banos (UPLB) for supporting my studies here in Australia.

CHAPTER 1

INTRODUCTION

Like any other piece of software, a compiler has certain attributes which make it desirable. One of these many attributes is portability. Portability is that property of a piece of software which enables itself to be moved from one computer system to another with significantly less effort than that of rewriting it.

Several methods of achieving portability in PASCAL have been developed since the first PASCAL compiler was written [Lecarme and Peyrolle-Thomas, 1978]. One of these, which is now considered fairly standard, involves the compilation into an "intermediate code". This intermediate code can be conceived as being the "assembly language" of a hypothetical stack-oriented machine.

There are several reported variants of this approach. One variant is the use of the PASCAL pseudocode (P-code) as an intermediate code. P-code may then serve as an assembly language to a hypothetical stack-oriented machine, P-machine, which executes P-code interpretively. Since P-code is interpreted, run-time efficiency in this approach is sacrificed for portability. Several implementations now developed use this approach. The first was the PASCAL-P implementation at Zurich in 1973-74 [Nori, et.al.,1975]. Another variant is the PASCAL-J system [Colemann, et. al., 1974]; the intermediate code used is the "universal intermediate code, Janus". In this system, the intermediate code is further translated to the assembly language of the machine on which the system is to run.

More recently, interest has been in an approach of implementing a group of languages, PASCAL included, on a collection of machines having one front end per language and one back end per machine. The method works by having each front end translate from its source language to a common intermediate code and each back end translate from the common intermediate code to a specific target machine's assembly language. In this way, the compiler is composed only of a front end and a back end. Transfering the compiler then from a host machine, A, to a target machine, B, would literally mean transfering only the back end part of the compiler to the target machine, B. The front end could be written in the language it compiles and translated to the common intermediate code in the host machine, A. With the back end part running in the target machine, B, one can have the whole compiler by running the intermediate code of the front end (produced in A) in the target machine, B.

The approach was considered in the design of a simple stack machine EM [Tanenbaum,et.al.,1982]. Front ends for EM includes ADA (subset), ALGOL 68 (in progress), BASIC (final test), C, PASCAL, and PLAIN. Back ends includes Intel 8080 (in progress) and 8086/8088, Zilog Z80 (in progress) and Z8000 (final test), Motorola 6809 (in progress) and 68000, DEC PDP-11 and VAX, and National Semiconductor 16032 (in progress). The same method has been successfully used in the portability of BCPL with the use of a Stack Language for Intermediate Machines (SLIM) [Fox,1978]. SLIM, unlike EM, was designed specifically for the programming language BCPL as front end. It has already been implemented in several back ends. Back ends for SLIM includes IBM 370 (Amdahl 470), PDP-11, Data General Nova (and Eclipse), Prime 750, Zilog Z80, Intel 8088 (or 8086), Perkin Elmer 3230, and Motorola 68000.

To take advantage of the fact that several back ends exist for SLIM, one of the purposes of this endeavor is to study the feasibility of other high-level languages, specifically PASCAL, as front ends to SLIM. Since SLIM was designed specifically for BCPL, this raises several questions. Some of these questions are "How compatible are BCPL and PASCAL?", "Will the present SLIM instructions be enough for PASCAL to be translated to SLIM?", "And if the SLIM instructions are not enough, will it be possible to extend SLIM without affecting its original design objectives?" These questions and others similar in nature were answered by translating the features supported by PASCAL-S and procedures as parameters to SLIM. The translations of the said features will be discussed in the succeeding chapters.

The choice of PASCAL from among the existing high-level languages, aside from BCPL of course, in the study of the possibility of SLIM for other front ends was due to several reasons. One is that PASCAL is extensively used for teaching programming [Lecarme,1974] and for writing portable software [Lecarme,1977]. Further, PASCAL is a machine-independent language, at least, much more compared to FORTRAN and PL/I. Moreover, PASCAL is a powerful language. It supports records, pointers, multidimensional arrays, and many other important structures. Finally, PASCAL is a simple language to implement. A one-pass compiler could be written for the language, compared to ALGOL 68 for which a multi-pass compiler is required (except Malvern ALGOL 68 compiler which is one-pass). Other reasons not mentioned will become apparent in later chapters.

CHAPTER 2

PASCAL AND ITS IMPLEMENTATIONS

Portability has been claimed to be a feature of both the PASCAL language and its compiler. This property of PASCAL makes it available in many machines. With the many existing implementations, several methods have been employed due to the wide differences of host machines. Methods used to implement PASCAL were so varied that several approaches have been used even for one particular machine. Although implementations differ from one machine to another and even for the same machine, the implementation methods used could be divided into three general methods of implementation: cross-compilers, PASCAL-P implementations, and self-compiling compilers. These general classes of implementations are not mutually exclusive and thus it is common to see an implementation which is a combination of the three general methods mentioned. All these approaches were a product of the desire to make PASCAL available from one machine to another with insignificant changes to the original compiler and with essentially the same run-time efficiency.

The succeeding sections of this chapter will describe in detail each general method of implementing PASCAL.

2.1 Cross-compiler method

Of the three general approaches of implementing PASCAL, the cross-compiler approach is the most popular implementation method used by microcomputer implementors. It has been used to implement PASCAL on minicomputers but more extensively on microcomputers. In this method, the

compiler runs on the host machine, A, and generates code for the target machine, B. Usually, the target machine is a microprocessor. An example implementation which used this method is described in Parson's paper "A Microcomputer Pascal Cross Compiler" [Parsons, 1978].

2.2 PASCAL-P Implementation

The next method commonly used by implementors is through the use of an intermediate code, PASCAL pseudocode (P-code). This method is widely known as the Pascal-P implementation. The compiler is made to generate PASCAL' pseudocode (P-code). Two methods are available for executing P-code.

The first method of executing P-code is by macro expansion. In this approach, each P-code is expanded to its assembler equivalent. The expansion of one P-code instruction grows to an undesirable size as the complexity of the P-code generated increases. Berry [Berry, 1978] reported that the total number of machine instruction in a decode segment of the interpreter for an integer "add" and an integer "subtract" is thirteen and a typical "load" and "store" can be accomplished in between twenty and twenty five machine instructions. This is the reason why macro expanding each P-code to their assembler equivalents is not usually used to execute P-code. However, there are some variants of P-code which can be translated to assembly language of the target machine. These variants are lower in level compared to the original P-code which was used in the first PASCAL-P compiler.

The second method is by writing a P-code interpreter. In this way, a user could provide himself a way to execute the program interpretively. There are several ways of writing a P-code interpreter. One way is to write

the interpreter in an assembly language. This way achieves run-time efficiency. Since writing the interpreter in assembly language will make it machine dependent, it is then obvious that the approach could achieve PASCAL portability up to P-code generation stage. Another common way of writing a P-code interpreter is to write it in the language (preferably a high-level language) in which the compiler is written, e.g., C. In this manner, PASCAL would be available on a machine which supports that high-level language. But, although in this way PASCAL portability is improved, run-time efficiency on the other hand would not be the same as when the interpreter is written in assembly language. In fact, writing the P-code interpreter in a high-level language may introduce an intolerable amount of run-time inefficiency.

Using a high-level language to write a PASCAL compiler and a P-code interpreter to achieve PASCAL portability raises several questions. It is a known fact that if the system is written in a high-level language, it can then be transfered to another machine that supports that high-level language. But, the big question is which high-level language to use. We should note that compilers for other high-level languages differ from machine to machine, too. This brings us to another method which is less machine dependent, the self-compilation method.

2.3 Self-compilation Method

As its name implies, the self-compilation method of implementing a language, requires that the compiler be written in the language it compiles. One obvious reason for doing so is that writing a compiler in a high-level language (PASCAL) is much easier than writing it in an assembly language. Problems arise when the language is not powerful enough for

compiler writing. In the case of PASCAL, however, this capability was considered in its design; thus it is possible to write a PASCAL compiler in PASCAL. The existence of several PASCAL compilers written in PASCAL [Welsh, 1977] is proof enough.

The main objective of the self-compilation method is to obtain a self-compiling compiler. The implementation method, therefore, is necessarily a bootstrap. Bootstrap methods could further be classified as half-bootstrap (pushing) and full-bootstrap (pulling).

The half-bootstrap method (pushing) is an implementation made on the target machine, B, with all the required tools on the host machine, A. One example of an implementation which adopted this method was the first PASCAL compiler for the ICL 1900 series, transported at Queen's University, Belfast, in 1971 [Welsh and Quinn, 1972]. The compiler was generated at Zurich and tested by a simulator of the ICL 1900 on the CDC 6000.

A more desirable but more difficult method is the use of a full-bootstrap (pulling). In this method, all work is done on the target machine, B. An implementor can achieve the transfer in three different ways. One approach is to rewrite the compiler in a high-level language available in the target machine, B. The resulting compiler, assumed to be inefficient, could be used to compile the compiler written in its own language. An example of an implementation that employed this method is the PASCAL implementation for the IBM 360-370 series made at Stanford University in 1972-74 [Russell and Sue, 1976]. The intermediary implementation language used was PL/I. The second approach is to write the compiler in the language it compiles. If an inefficient, or at worst unsatisfactory, PASCAL compiler exists in the the target machine, B, then

It is used to compile the compiler which is written in PASCAL. This approach was used in the second compiler developed for CDC 6000 series at Zurich Institute of Technology in 1972-74 [Jensen and Wirth, 1974]. The first compiler was completely rewritten and the first was used to compile the revised compiler. The third approach is to divide the compiler into two parts: the first part (written in the language it compiles) generates an intermediate code which is translated by the second part. This implementation method was first defined at the University of Colorado in 1975-76 [Coleman, et.al., 1974]. The first part of their compiler generates the universal intermediate language, Janus. Janus code may then be translated into a target machine's machine code by using a macroprocessor.

CHAPTER 3

THE SLIM MACHINE AND ITS ASSEMBLER

SLIM is a simple, one-accumulator, stack-oriented hypothetical machine. It was first proposed by Mark Fox [Fox, 1978], but was later enhanced by J. E. L. Peck. The machine was designed with the following principal aims [Peck, In preparation]:

1. To reflect current machine architecture, if possible;
2. To obtain a reasonably simple machine such that it can be used for teaching the elements of computing;
3. To obtain a machine that is suitable as target machine for high-level languages such as BCPL;
4. To obtain a tool for achieving portability of systems programs; and
5. To obtain a machine on which it is possible to have an operating systems .

This chapter will give a description of the SLIM machine and its assembler. The complete SLIM instruction set will also be given at the end of this chapter.

3.1 The SLIM Machine

SLIM is a machine very similar to a conventional computer in that it consists of a memory and a processor (see figure 3.1).

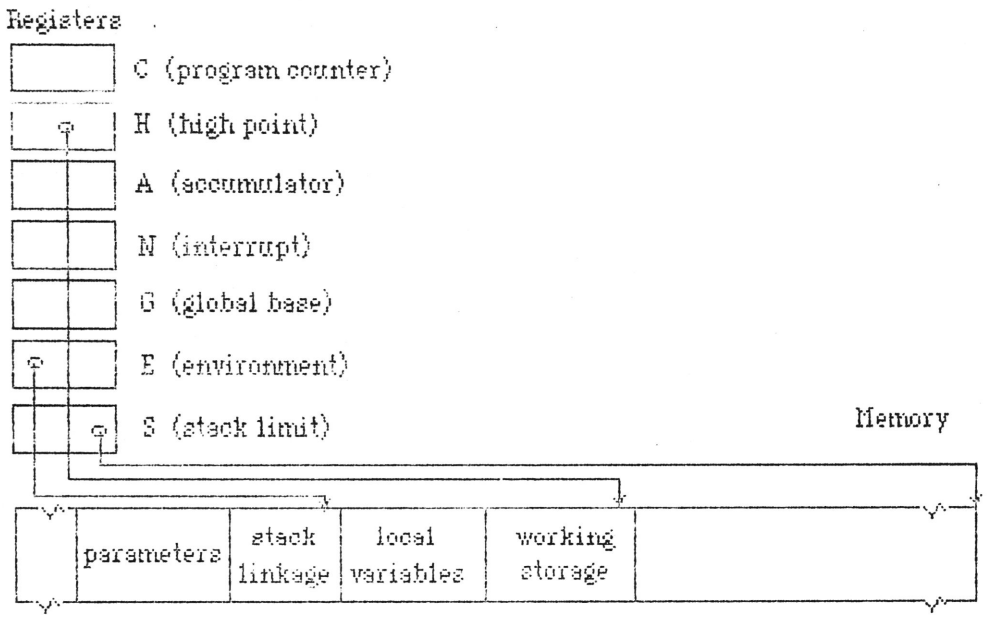

Fig. 3.1 The SLIM Machine

3.1.1 Memory

The memory of SLIM is a sequence of cells. Each cell contains 'n' bits with the value of 'n' implementation dependent. It may be 16 bits, 32 bits, or more, but the choice depends entirely on the number of bits required for an address on the target machine and the memory available in the target machine. The cells are addressed consecutively starting from 0 to 'm'. The value of 'm', as with the number of bits per cell, is machine dependent. However, the size of 32K cells was found to be a comfortable size for many SLIM implementations, e.g., on Perkin Elmer 3230, Motorola 68000, IBM 370 (Amdahl 470), etc.

3.1.2 Registers

SLIM has a total of seven (7) registers, but the three most important registers are the accumulator (A), the environment register (E), and the high point register (H).

The accumulator (A) is where all arithmetic and logical operations take place. The same register holds the value returned by a function.

The environment register (E) is used to hold a pointer to the environment, i.e., parameters and local variables of a procedure. All run-time linkage, parameters, and local variables are accessed through the environment register.

The high point register (H) points to the last useful cell on the stack' and is used to regain the stack space when the execution of a procedure is complete. Further, this register is used in pushing and popping values onto and from the stack.

The remaining registers are the program counter (C), which holds a pointer to the next instruction to be executed, the global register (G), which holds a pointer to the first cell of a sequence of cells reserved for BCPL global values, the interrupt register (N), and the stack limit register (S).

3.1.3 Instructions

Typical SLIM instructions may contain at most three fields - the operator, the operand modifier, and the raw operand. The operator field is always present in the instruction while the raw operand and the operand modifier may or may not be, depending on the type of instruction.

An operation may be L for load, i.e., move data from memory to accumulator (A), S for store, i.e., move data from accumulator (A) to

memory, + for add, J for jump, and so on.

A raw operand is either an unsigned or signed number, a character, the H register, or a label (@n where 'n' is an integer).

The remaining field is the operand modifier. It is used, if it is part of the instruction, to qualify the meaning of the raw operand. An operand modifier is either E (modified by environment), G (modified by global), I (modified by indirection), IE (combination of environment and indirection), or IG (combination of global and indirection).

Figure 3.2 shows the possible formats of a SLIM instruction.

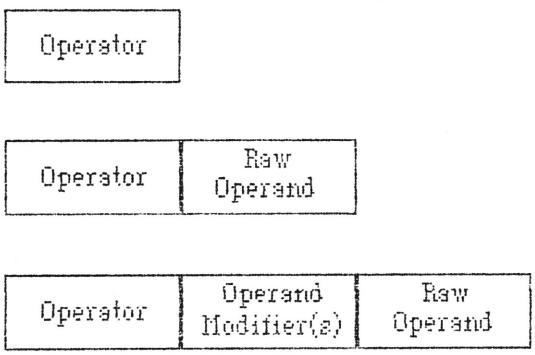

Fig. 3.2 Formats of SLIM instructions

Examples of complete SLIM instructions are:

Instruction	Action
L'a	Load the character 'a' into the accumulator
L@2	Load the address of label 2 into the accumulator

LIE2	Load the content of the cell into the accumulator, the cell being the second in the environment
P	Push the value of the accumulator into the stack after incrementing the value of the H register by one.

3.2 The SLIM Assembler

Unlike most of the conventional computer's assembly language, the SLIM assembler was designed to be compact. Compactness is illustrated by the fact that SLIM assembler uses a free format. It does not rely on spaces to delimit the fields of an instruction. It does allow any number of instructions per line and instructions can be written anywhere in the line. To illustrate this property of the SLIM assembler, an example of SLIM assembler and its equivalent in actual machine's assembly language, Motorola 68000, is given.

SLIM	M68000
@1: L12 +20	label1: move.l #12, d0 add.l #20, d0

3.2.1 Assembler Pragmats

Assembler pragmats are in some way the same as instructions, but do not cause any code to be generated. In SLIM, they are used to increase the readability of the program. In addition, they are used to give the programmer an idea of the code and data area of the program. We list all the assembler pragmats with their corresponding meaning. The functions of

the pragmats should be obvious from the meaning given.

Pragmat	Meaning
$$"program"	Start of section "program"
$"start"	Start of procedure "start"
$$	End of current section
$	End of current procedure
$:	End of code and start of data

The word "section" is used in the sense of BCPL.

3.2.2, Modifiers

As mentioned in the previous section, a SLIM instruction may or may not take an operand modifier. If it does, then the operand modifier together with the raw operand is used to calculate the operand (modified) that is finally used by the operation. For example, in the instruction L2, 2 is the raw operand and since the instruction does not have an operand modifier, the final operand used is 2. But in the instruction LE2, the E modifier is in the instruction, hence the final operand used by the operation is an operand that is calculated from the content of the E register and the raw operand 2. To illustrate how the calculation of the final or modified operand proceeds, the succeeding paragraphs will give a detailed description of how each of the modifiers is used to compute the final operand.

Firstly, we consider how the modified operand is computed for instructions with the E (environment) modifier. Computation of the final operand is simple: it is done by adding the content of the E register to the raw operand. Thus in the example given in the previous paragraph, if

the E register holds the value 040000, the final operand used by the instruction LE2 is 040002.

Similarly, for G (global) modifier, the final operand is computed by adding the content of the G register to the raw operand. It is worth noting that E and G modifiers may not both appear in one SLIM instruction and that they can not be used to modify an H raw operand. In consequence, the instructions LEG'0 and SEH are therefore not valid SLIM instructions.

The remaining modifier is the I (indirection) modifier. It can be used independently, e.g., SI@2, or together with the E modifier, e.g., +IE2, or with the G modifier, e.g., CIG10. The final operand is obtained by first adding the content of the register specified by the other modifier, i.e., E or G, if one is present of course, to the raw operand, then taking the result as an address of the cell which contains the final operand. For example, if the E register contains 010000, the final operand then of the instruction LIE4 is whatever value is stored in the cell whose address is 010004.

3.2.3 Operands

Every operand is either a number, a character, the letter H, a label, or a string.

If the operand is a number, then it may be a signed or unsigned number, e.g., LIE7 and LIE-7. Further, the number may be an integer or a real number, e.g., L1 and L1.25. Moreover, it could be in decimal form (e.g., L16), octal form (e.g., LO20) or hexadecimal form (e.g., LX10).

If the operand is a character, a single quote (') precedes the character operand, e.g., L'c. What is actually manipulated by the machine is the integer representation of the character which requires only a byte of storage. Special characters are preceded by an asterisk and follow the escape rules of BCPL, e.g., L'*N for newline, L'*S for space character, etc. Of course, the integer representation of the character could also be used, i.e., in ASCII, L10 for newline, L32 for space character, L69 for character E, etc.

In the case where the operand is the character H, as pointed out earlier, it can not be accompanied by an E or a G modifier. The value of the operand when H is used is whatever is on top of stack. The value is then popped from the stack.

If the operand is a label (@n), the operand is actually an address in the program area which is supplied by the assembler. Hence, it is no different from an integer operand, except that the value may change from one execution to another depending on where the program is loaded in memory.

There is only one instance where the string operand is used, i.e., in the storage reservation (pseudo)instruction (D). It occurs in the form of a BCPL string, i.e., "string". An example of an (pseudo)instruction which uses such an operand is D"string". A (pseudo)instruction like D"string" actually reserves 7 bytes of storage, the first byte being the length of the string.

3.2.4 Operators

The operations supported by SLIM could be classified into 7 categories: load and store operators, stack operators, program control or jump operators, accumulator operators, logical or bitwise operators, arithmetic operators, shift operators. Others not falling in any of the category given will also be discussed under the heading, miscellaneous operators.

3.2.4.1 Load and Store Operators

The most elementary, but important, operations in the SLIM instruction set are the L (load) and S (store). L moves data to the accumulator and S moves it from the accumulator.

The basic L operator may or may not take an operand modifier. If it does not, then it is used to move a constant into the accumulator. In this case, the instruction is similar to most actual machines' load immediate instruction. An example is L100 which sets the value of the accumulator to 100. Or, it may take an operand modifier which could be of the form: E, G, I, IE, or IG. The table below summarizes the effects of a load instruction for each of the operand modifiers and the raw operand 10.

Instruction	Action
LE10	Load the address of the cell whose address is the value in the E register plus 10
LG10	Load the address of the cell whose address is the value in

	the G register plus 10
LI10	Load the content of the cell whose address is 10
LIE10	Load the content of the cell whose address is the value in the E register plus 10
LIG10	Load the content of the cell whose address is the value in the G register plus 10

Similarly, the S operator has several forms. It can also take an integer raw operand without any operand modifier, e.g., S100. An instruction of this form does have a meaning, i.e., S100 means store the content of the accumulator in a cell whose address is 100, but this usually is not generated by a compiler which uses SLIM as a target machine. It can also take an operand modifier which could be of the form: E, G, I, IE, IG. Again, the table below summarizes the meaning of the instruction S with each of the operand modifiers and raw operand 10.

Instruction	Action
SE10	Store the value of the accumulator in the cell whose address is the value in the E register plus 10
SG10	Store the value of the accumulator in the cell whose address is the value in the G register plus 10
SI10	Store the value of the accumulator in the cell whose

	address is the content of the cell whose address is 10
SIE10	Store the value of the accumulator in the cell whose address is the content of the cell whose address is the value in the E register plus 10
SIG10	The same as SIE10, except that the register used is G

Theoretically, the forms of the basic load and store instructions given above are legal SLIM instructions. But many of them are not generated by a compiler, e.g., SIE10, SI10, LG10, etc.

There are a number of variants on the basic L and S operators.

The first variants are the L! (load subscripted cell) and S! (store subscripted cell). These variants allow SLIM to be able to manipulate the elements of a vector (array). For example, L!10 will load the value of the tenth cell beyond the cell whose address is in the accumulator prior to this instruction. So if the accumumulator contains 01000, the instruction will load into the accumulator the content of the cell whose address is 01010. Similarly, S!10 will store the value that is currently on top of stack to the cell whose address is 01010 and decrement the H register by 1.

The second variants of the load and store operators are the L% (load byte) and S% (store byte). These variants were specifically introduced in SLIM to help manipulate strings. The action of these two instructions are similar to the first variant (subscripted cell), except that the operand is

considered as byte offset instead of cell offset. Given the same value in the accumulator, i.e., 01000, the instruction L%10 will load the byte pointed to by the corresponding actual machine address of the cell whose address is 01000 plus 10 bytes. So if the corresponding machine address of cell 01000 is 012344, then the content of byte 012354 will be loaded into the accumulator. Since the size is a byte, the value will occupy the eight rightmost bits of the accumulator and the other bits are set to zero.

The third variants of the basic L and S operators are the L: (load field) and S: (store field) operators. These instructions are used to manipulate fields of bits but will not be discussed further.

Lastly, the variants L$ (load device) and S$ (store device) operators are used primarily for the implementation of I/O operations.

3.2.4.2 Stack Operators

When a procedure is declared, most often it declares several variables local to itself. Once declared, these variables have to be allocated space in the stack. During execution, the stack may also be used as temporary storage of some intermediate results of a computation. When the computation is complete, the space occupied by these intermediate results must be deallocated to save stack space. The operators that perform allocation and deallocation of stack spaces are known in SLIM as stack operators. SLIM provides three stack operators: P (push), PL (push load), and M (modify). A pop operation is implied by any instruction whose operand is M.

The P (push) operator takes no operand. Its effect is to copy the

content of the accumulator to the top of the stack, after incrementing the H register by one. The instruction preserves the value in the accumulator. It is this instruction that is mainly used to allocate space to data structures which require one cell of storage.

The second stack operator, PL (push load), takes the same operand as the L operator. It first copies the content of the accumulator to the top of the stack, of course after incrementing the H register. The loading into the accumulator follows. One important observation about this instruction is that the two instructions P L do not always have the same effect as PL. The difference happens when the load part takes on the character H as the raw operand: PLH interchanges the values of the top of stack and the accumulator while P LH pushes the value into the top of stack and pops it back into the accumulator. The total effect of P LH is nil.

The M (modify) operator is the last of the three stack operators. Its action is to add whatever is the value of its operand to the H register. This operator is used by SLIM for allocating space to data structures that require more than one cell of storage. Further, it is very useful in deallocating unused space in the stack.

3.2.4.3 Program Control or Jump Operators

SLIM provides unconditional jump (J) and conditional jumps (T and F) to change the sequence of execution of the program. It is, naturally, important to be able to jump to another part of the program.

The action of the unconditional jump (J) is to replace the value of the C (program counter) register by the value of its operand (modified if

necessary). Similarly, conditional jumps (T and F) follow the same action except that the action is done only after the value in the accumulator is found to be an appropriate value. This means that the jump will occur only when the value in the accumulator is -1 (true), for jump if true (T) or when the value is 0 (false), for jump if false (F).

3.2.4.4 Accumulator Operators

A monadic operator in most high-level languages is called an accumulator operator in SLIM. It takes no operand and its sole function is to alter the value in the accumulator. Monadic operators are the ~ (complement) operator which computes the complement of the accumulator and returns the result in the accumulator, the - (negate) operator which negates the value in the accumulator, and the | (absolute) operator which takes the absolute value of the accumulator. Below is a table showing the effect of the operation on the accumulator (assuming a two's complement machine).

Operator	Accumulator	
	Before	After
~ (complement)	-1	0
- (negate)	-1	1
| (absolute)	-1, 1	1, 1

3.2.4.5 Comparison Operators

A set of SLIM operators that take one operand and yield a boolean value in the accumulator are called comparison operators. The actual action of a comparison operator is to compare the value in the accumulator with its operand. After the comparison, the value of the accumulator is replaced

by the result of the comparison, i.e., 0 for false result and -1 for true result. The comparison operators are the = (equal) operator, ~= (not equal) operator, < (less than) operator, <= (less than or equal to) operator, > (greater than) operator, and >= (greater than or equal to) operator.

3.2.4.6 Logical or Bitwise Operators

The action taken by this set of operators, logical or bitwise operators, is the same as that of the comparison operators except that comparison is done bit by bit and the result is returned in the corresponding bit of the accumulator. The logical operators are the /\ (logical and) operator, \/ (logical or) operator, == (equivalent) operator, and the ~~ (not equivalent) operator. Below is a summary of results expected for all possible combination of bits.

Accumulator Bit	Operand Bit	Operator /\	\/	==	~~
0	0	0	0	1	0
0	1	0	1	0	1
1	0	0	1	0	1
1	1	1	1	1	0

3.2.4.7 Shift Operators

We have seen, in the previous set of operators, how individual bits are manipulated in place. The next set of operators, shift operators, move bit patterns within the A register. SLIM provides only two instructions for doing this: one moves the bit pattern to the left (<<) and the other moves it to the right (>>). The operator takes an operand which determines the number of bits the pattern will be shifted. For example, suppose the

accumulator contains 16 (in binary, 10000), the instruction <<2 will leave the value 64 (in binary, 1000000) in the accumulator and >>2 will leave 4 (in binary, 100) in the accumulator.

3.2.4.8 Arithmetic Operators

We have so far discussed a wide range of SLIM instructions, but have not mentioned anything about performing arithmetic. SLIM like any other conventional machine provides a way of doing addition (+), subtraction (-), multiplication (*), division (/), and taking remainder (/*). The action of the operator is to perform the required arithmetic (depending on the type of operator of course) on the accumulator and the operand. The result of the operation is delivered to the accumulator. The order of operands in the operation is, accumulator operator operand. This order is important because not all operations are commutative. The table below shows the result of doing the operation when the accumulator holds the value 10 and the operand of the instruction is 3.

Operator	Accumulator	
	Before	After
+3	10	13
-3	10	7
*3	10	30
/3	10	3
/*3	10	1

The same operators, except /*, can also be used for operands with real value, but the operator must be qualified by prefixing '#' before it. For example, adding 1.25 to the accumulator would mean generating the code #+1.25.

3.2.4.9 Miscellaneous Operators

Some operators do not fall into any of the categories we have discussed. These operators are incorporated in the SLIM instruction set to facilitate translation of high-level languages into SLIM. Further, most of these instructions do not have an equivalent one instruction in almost all actual machines' instruction set, but they could be simulated using several of the actual machine's instructions.

3.2.4.9.1 Procedure Call (C) and Return (R)

The first two and most important of these operators are the C (procedure call) and R (procedure return) operators. The main purpose of the operators C and R is to facilitate the calling of procedures in order to give SLIM the power of a high-level language type of procedure call.

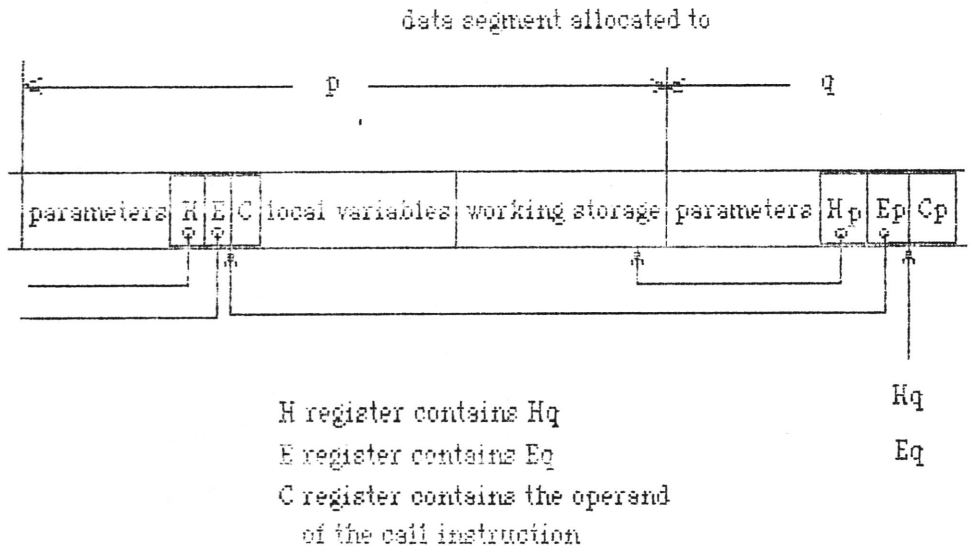

Fig. 3.3 The state of the stack after the the execution of a call to procedure 'q' by p.

The C (procedure call) operator may or may not take an operand modifier. The final operand, obviously, is the entry point of the called procedure. The C operator uses some information supplied by previous instructions (parameters) and the data (D) instruction that always follows the C instruction. Examples of this call are L1 PL10 C@2 D4, CI@2 D0, etc. The operand of the D instruction supplies the number of actual parameters. We should note that when a procedure call is made, the machine executing the program should make arrangements to return control to the proper part of the program once the execution of the procedure is complete. In SLIM, it is this C operator which makes the arrangements.

First, the C operator processes the parameters passed to the procedure by pushing the last parameter from the accumulator to the top of stack, of course only if the procedure requires at least one parameter, i.e., when the operand of the D (pseudo)instruction following the C instruction is greater than 0. Moreover, if the number of formal parameters is not the same as the actual parameters, the C operator allocates space if necessary. The number of formal parameters is supplied by the first data (D) (pseudo)instruction of the called procedure. The code for a procedure always starts with this D (pseudo)instruction whose operand is the number of formal parameters.

Once the parameters are in the stack, the C operator sets up the SLIM link. The SLIM link follows the parameters in the stack and this is where information, necessary to gain control after completed execution of the procedure, is stored. To be specific, the SLIM link stores the environment (E), high point (H) register, and the return address, currently

held in the C register, of the calling procedure.

Lastly, the C operator transfers program control to the called procedure by placing the entry point of the called procedure into the C register.

It is instructive to look at the state of the stack when a procedure, p, calls another procedure, q. Figure 3.3 shows a snapshot of the stack immediately after the operator C (procedure call) is executed.

The R operator has a simpler action compared to the C operator. It moves the data stored in the SLIM link into their corresponding registers and tranfers control to the instruction after the procedure call. Note that the stack space consumed by the called procedure is automatically deallocated after R has completed its action. A snapshot of the stack after a return from procedure, q, is complete, is shown in figure 3.4.

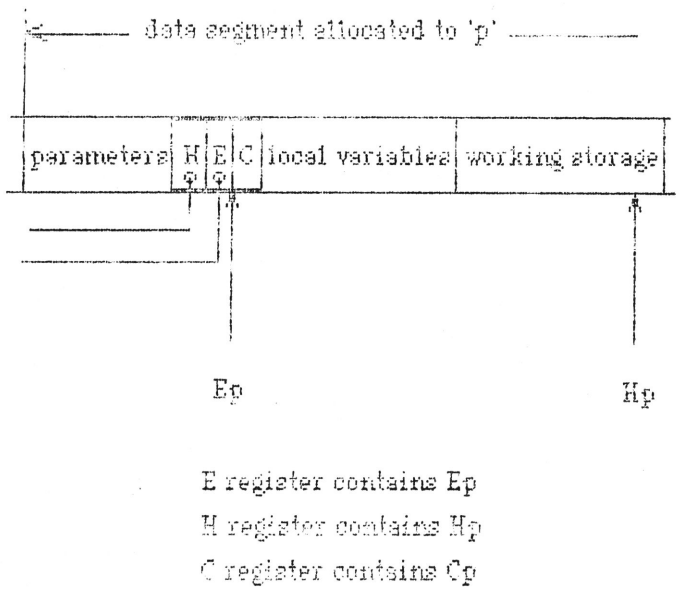

Fig. 3.4 The state of the stack after a return from procedure 'q'.

3.2.4.9.2 Sequential (?S) and Indexed (?I) Switchon Operators

A pair of operators each of which does not have an equivalent one instruction in actual machine's instruction is the sequential (?S) and the indexed (?I) switchon. These are SLIM instructions which correspond to CASE-type statements of most high-level languages, e.g., CASE statement of PASCAL, SWITCH statement of C, and SWITCHON statement of BCPL. One operator is actually sufficient, but two separate operators were provided to take advantage of the nature of the case values. Sequential (?S) switchon is designed to work efficiently on case values which differ by more than one while' indexed (?I) switchon is primarily suited for case values which differ by one.

3.2.4.9.3 Exchange (X) Operator

The exchange (X) instruction was found to be necessary for translating BCPL's updating operator, i.e., +:=, -:=, etc. It has the effect of swapping the value of the accumulator and that of the cell below the top of the stack.

3.3 The Complete SLIM Instruction Set

In summary, we list all the SLIM operators in the table below. The table gives a brief description of the operator, and the microcode for each operator to help visualize its actions. The microcode is written in BCPL. The table does not show all the possible operands and operand modifiers; instead we represent the operand, modified or unmodified, by the letter W. The same table appears in Peck's "The Essence of Portable Programming."

SLIM Instructions

Instruction	Mnemonic	Microcode
Load cell	LW	A := W
Store cell	SW	!W := A
Load cell subscripted	L!W	A := A!W
Store cell subscripted	S!W	LET V = !H; H -:= 1; A!W := V
Load byte	L%W	A := A%W
Store byte	S%W	LET V = !H; H -:= 1; A%W := V
Load field	L:W	A := W of A
Store field	S:W	LET V = !H; H -:= 1; W OF A := V
Load device	L$r	A := A!r
Store device	S$r	LET V = !H; H -:= 1; A!r := V
Push and load cell	PLW	H +:= 1; !H := A; A := W
Jump	JW	C := W
True jump	TW	IF A = TRUE THEN C := W
False jump	FW	IF A = FALSE THEN C := W
Modify high point	MW	H +:= W
<dop>**	<dop>W	A := A <op> W
<mop>*	<mop>	A := <mop> A
Procedure call***	CW	
Procedure return	R	C := E!0; H := E!(-2); E := E!(-1)
Push	P	H +:= 1; !H := A
Exchange	X	W := H!(-1); H!(-1) := A; A := W
Originate***	O	
Void	V	no operation
Quit	Q	exit
Switchon sequential***	?S	
Switchon indexed***	?I	
Non-local access***	U	

*** microcode for C, O, ?S, ?I, U are given in Peck's "The Essence of Portable Programming"
** <dop> stands for dyadic SLIM operators
* <mop> stands for monadic SLIM operators

Dyadic and Monadic SLIM Instructions

Instruction	Mnemonic	Microcode
Dyadic Instructions		
Add*	+W	A := A + W
Subtract*	-W	A := A - W
Multiply*	*W	A := A * W
Divide*	/W	A := A / W
Remainder	/*W	A := A REM W
Equal to*	=W	A := A = W

Not equal to*	~=W	A := A ~= W
Less than*	<W	A := A < W
Less than or equal to*	<=W	A := A <= W
Greater than*	>W	A := A > W
Greater than or equal to*	>=W	A := A >= W
Equivalent	==W	A := A == W
Not equivalent	~~W	A := A ~~W
Logical and	/\W	A := A /\ W
Logical or	\/W	A := A \/ W
Right shift	>>W	A := A >> W
Left shift	<<W	A := A << W
Monadic Instructions		
Complement	~	A := ~A
Negate*	-	A := -A
Absolute*	|	A := ABS A
Float	#:	A := FLOAT A
Fix	#.	A := FIX A

)

* Instructions with floating point equivalents: #+, #-, #*, #/, #=
#~=, #<, #<=, #>, #>=, #-, #|

CHAPTER 4

EXTENSIONS TO SLIM FOR PASCAL AND PASCAL-LIKE LANGUAGES

Since SLIM was designed for BCPL, features supported in PASCAL but not in BCPL is the reason for these extensions. The answer to the question "Will the existing SLIM instructions be enough for PASCAL to be translated to it?" is obviously "no". But with several incompatibilities between PASCAL and BCPL, the existing SLIM instruction is enough to handle the features of PASCAL except for non-local variables.

The succeeding sections will illustrate why SLIM is not capable of handling non-local variables and the necessary extensions for it to be a suitable intermediate language for languages that support non-local variables.

4.1 Environment of a Procedure

When a procedure is activated, a data segment is allocated to it in the stack. This data segment may contain the parameters passed to the procedure, variables local to itself, working storage, and the stack link. The stack link, of course, is necessary to restore the stack to its original state when the execution of the called procedure is complete. We refer to the original state of the stack as the state of the stack just before the call to a procedure is made. It is this data segment plus other data segments allocated to procedures activated before it, which form the environment of a procedure. To be precise, the environment of a procedure is the set of data segments in the stack that it can access.

In languages like BCPL, environment means the data segment allocated to the procedure. But in most block-structured languages like PASCAL, ALGOL 60, ALGOL 68, PL/I, etc., environment means the segment allocated to the procedure plus those data segments allocated to procedures activated before it, which could be accessed by the procedure according to the scope rules of the language. The reason for this difference of extent of environment of a procedure is simple: BCPL does not allow non-local variables. These non-local variables are declared outside the procedure where they are accessed.

The consequence of BCPL's not supporting non-local variables is that its stack link is simple. Since the environment of a procedure is only the stack segment allocated to it, all that is needed in the link is a chain which connects the newly activated procedure to the old segment. This function, if you noticed in the actions of operator C (procedure call), is handled by keeping the content of the E register in the stack link once a new procedure is activated. Hence, when SLIM is used for BCPL, the E register does have a dual function, to chain the data segments and at the same time represent the environment of the procedure. But for languages that support non-local variables, or at worst non-local variables plus procedures as parameters, the E register is not enough to describe the environment of a procedure.

As mentioned, one of the functions of the E register is to chain one data segment to its immediate predecessor in the stack. This chain is what is commonly known, in compiler implementations, as the dynamic link. To access variables though, local and non-local, two methods are available. One is the use of the display method, but this is ruled out immediately because it would mean an addition of registers equal to the number of

nesting levels allowed. This number is implementation dependent. But, usually from 5 to 7 nesting levels is used. The other method is to establish another chain, the static chain, which connects those data segments accessible to the currently executing procedure. The static link method requires only the additional of one register (U) to the existing SLIM registers, and one operator, the U operator.

4.2 Environment of a PASCAL-like Procedure

The U register and the E register together represent the environment of a procedure in languages that support non-local variables. The U register points to the nearest segment accessible to the procedure. As a consequence of the introduction of the U register, the SLIM link for languages like PASCAL is composed of at least 4 cells compared to at least 3 cells for BCPL (see figure 4.1).

SLIM link for PASCAL SLIM link for BCPL

Fig. 4.1 Possible SLIM link for PASCAL-like languages and BCPL-like languages

The U entry in the link holds a pointer to the nearest accessible segment. In turn the U entry of that segment points to the next nearest accessible segment, and so on. This relationship between the U entry of one link to another will become clear after considering the example below. The example takes a specific PASCAL program and shows the stack at some instant of execution.

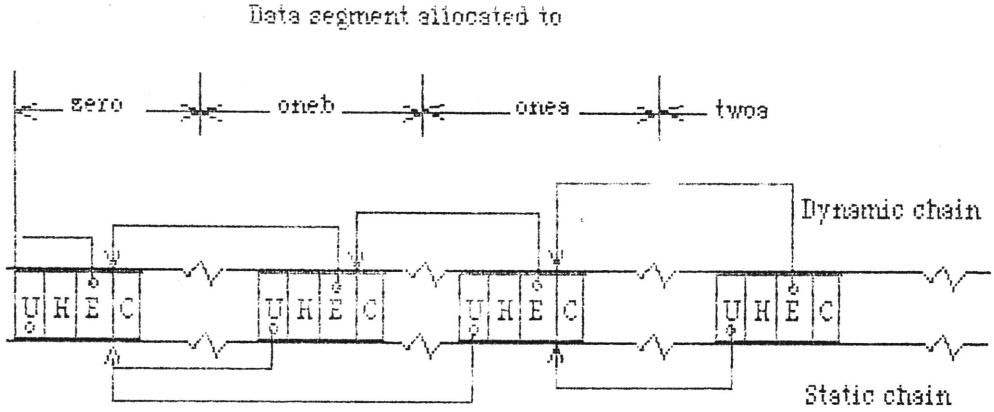

Fig. 4.2 The stack showing the dynamic and static links.

Example:

```
program zero(output);
 procedure onea();
   procedure twoa();
     begin end;
   begin  twoa end;
 procedure oneb();
  begin onea end;
 begin oneb end.
```

Figure 4.2 shows the state of the stack when the execution is in procedure twoa, i.e., zero --> oneb --> onea --> twoa. The chain at the top of the stack is the dynamic chain and the one at the bottom of the stack is the static chain. The stack grows from left to right in this example.

SLIM avoids the overhead of computing the value of the static link, by requiring the user to establish it himself. The value of the U entry in the link could be set by passing it as one of the parameters. This method of establishing the static chain does not introduce any overhead and it does not pose any problem to the code generator because the value of the static link is actually known at compile time. One example of a call that

sets the value of the static link is LEO C@9. The LEO instruction does set the cell reserved for the U register.

Now that the static link is established, it is possible to access variables declared outside the procedure, provided of course the access obeys the scope rules of the language. The extended SLIM provides another operator, the U operator, for doing just that.

The action of the U operator is to specify which E register to use from among those stored in the stack. The operand of the U instruction suggests that in the next occurence of an E modifier, it has to follow the static link down by the number of levels equal to the operand of the U instruction and use the E register in that segment. Usually, the U instruction comes before an instruction which takes the E modifier. For example, U2 LIE3 is an instruction for accessing a non-local variable declared two levels shallower than the procedure where it occurs. Specifically, the action of the two instructions is to load a value into the accumulator, but in the computation of the final operand of the load instruction, the E register used is that stored two levels down the static chain. Accessing local variables on the other hand can be done by instructions like U0 LIE3. The instruction U0 suggests the use of the E register of the currently executing procedure. But usually, U0 could very well be omitted and the same action will be taken, i.e., the action of U0 LIE3 is the same as LIE3.

4.3 Procedures as Parameters

Allowing non-local variables in a language will almost certainly create difficulty to implementors. This is because the representation of

the procedure's environment can not be handled by the dynamic link alone. The difficulty is worse, if aside from non-local variables, the language allows procedures as parameters. This will create the problem of how to set its local environment. We mentioned that in SLIM, the local environment of a procedure in a language that supports non-local variables is set by the user. But, how about those procedures which are passed as parameters? Will their local environment be set in the same manner?

First, we should observe that if the user is required to set the local environment of a procedure, the said local environment should be known at compile time. This will depend on whether deep (static) or shallow (dynamic) binding of procedure references is used by the language being translated.

When shallow (dynamic) binding of procedure references is used, the environment of a procedure is not set until the procedure is called. This binding is used by languages like APL, LISP, and SNOBOL.

In the case of PASCAL and PASCAL-like languages, deep (static) binding is used. In deep binding, procedure identifiers are statically bound to their names. This suggests that the environment of a procedure is fixed and known at compile time. Therefore, arrangements in SLIM where the user is required to set the environment of a procedure himself is safe.

The succeeding paragraphs will concentrate on how SLIM can be used to set the environment of procedures that are passed as parameters in a language that uses deep binding of procedure identifiers to their names.

The solution adopted by SLIM, as mentioned, is to require the SLIM

user, i.e., the compiler writer, to set the environment of a procedure himself. Since the environment of a procedure is fixed, when passing a procedure as a parameter, the user is required to pass the procedure's local environment together with its entry point. This way, when a procedure is called, the local environment used is the one that was passed instead of the local environment derived from the local environment of its caller. To illustrate this point, we consider an example program and its translation. The translation assumes a 5-cell SLIM link.

Example:

PASCAL	SLIM
program passprocedure(output);	$$"passprocedure" J@2
	@1: DO M1 J@3
procedure p(procedure r);	$"p"
	@4: D2 M1
begin	
r	@5: LIE-5 CIE-6 DX8000
end;	R $: $
procedure q;	$"q"
	@6: DO M1
begin	
end;	@7: R $: $
begin	
p(q)	@3: LI@8 P UO LEO P UO LEO
	C@4 DX8002
end.	R $:
	@8: D@6 $$
	@2: L@1 SG1 .

Observe the translation of parameter 'q' in the call to procedure 'p'. Since 'q' is a procedure, when it is passed as a parameter, its entry point, i.e., @6:, and its local environment is passed to the called

procedure by the instructions 'LI@8 P U0 LE0'.

Another important observation is how the call to a procedure which was passed as a parameter, i.e., call to procedure 'r', is translated. Note that the local environment is set by simply accessing its value from the parameter area, i.e., LIE-5, instead of deriving it from the local environment of the current procedure, e.g., U0 LE0.

CHAPTER 5

THE PASCAL TRANSLATOR

A translator is a program that takes as input a program written in one programming language, the source language, and yields as output a program in another language, the object language. The object language may be relocatable object code, an assembler language of an actual machine, an intermediate language (assembler language of a hypothetical machine) or another high-level language. Regardless of the nature of the source and target languages, all translators have one universal purpose. This purpose is to translate a piece of code of the source language to an equivalent code of the target language such that when executed by the target machine, the actions will be as specified by the source code.

In this chapter, we shall be discussing the design of the translator which was constructed to translate the high-level language PASCAL (source language) to SLIM (object language). In addition, some of the techniques and methods used in the translation will also be discussed.

4.1 Design of the Translator

The quality of the object code generated by a translator is mainly dependent on the translator's design objectives. Of course, generating the most efficient object code is of importance but achievement of this will depend on how the translator is written. The following are the design aims of the translator which was constructed to translate PASCAL to SLIM:

1. To minimize the time taken to compile source

programs;

2. To produce a reliable and reasonably (if not the most) efficient object code; and

3. To keep the translator as simple as possible.

Unfortunately, these aims are to some extent conflicting. Obviously, a translator will have to take a little longer and be bigger if it has to generate efficient object code. But, as will be discussed in the chapter on code improvement, the technique used to improve the code is simple enough such that the increase in time to compile and the increase in the size of the translator are negligible.

In keeping with the aims (1) and (3), the translator was constructed such that it translates PASCAL to SLIM in one pass. The translation takes place while a PASCAL source program is being read in to the computer. The object code, SLIM code, is generated and is ready for further translation to an actual machine's assembly language as soon as the reading of the PASCAL source program is complete.

Reliability of the object code should be the primary aim of every translator. Generating a not so efficient but reliable object code is certainly better than attempting to generate the most efficient object code which may not be correct all the time. In this respect, the translator was constructed in such a manner that it translates the program by decomposing it into simple parts and replacing these simple parts with their correct equivalent object codes.

4.2 Program Structure of the Translator

As mentioned earlier, the translator is a one-pass translator. It obtains its input through a source-oriented scanner. The scanner returns PASCAL lexemes (tokens) which serve as input to the analyzer. The said analyzer is based on a bottom-up parsing principle. These same methods were used in the Elliot ALGOL translator and the first PASCAL compiler [Wirth, 1977].

The code generator part of the translator generates object code while the analyser is busy analysing the program. It does not maintain extra data structures like translator stacks and parse trees. The code generator employs a tree-walking technique to achieved the translation. The tree-walking technique will be discussed in detail in the next section.

The translator as a whole is composed of a number of mutually recursive procedures. Each procedure is capable of translating one kind of source program fragment and each procedure can generate a variety of different kinds of object code fragments depending on the structure of the source code fragment presented to it.

To visualize how the procedures of the translator are related to each other, we show in figure 5.1 the dependence diagram of the procedures in the translator.

The translator supports only those features supported by PASCAL-S and procedures as parameters. It is written in C language.

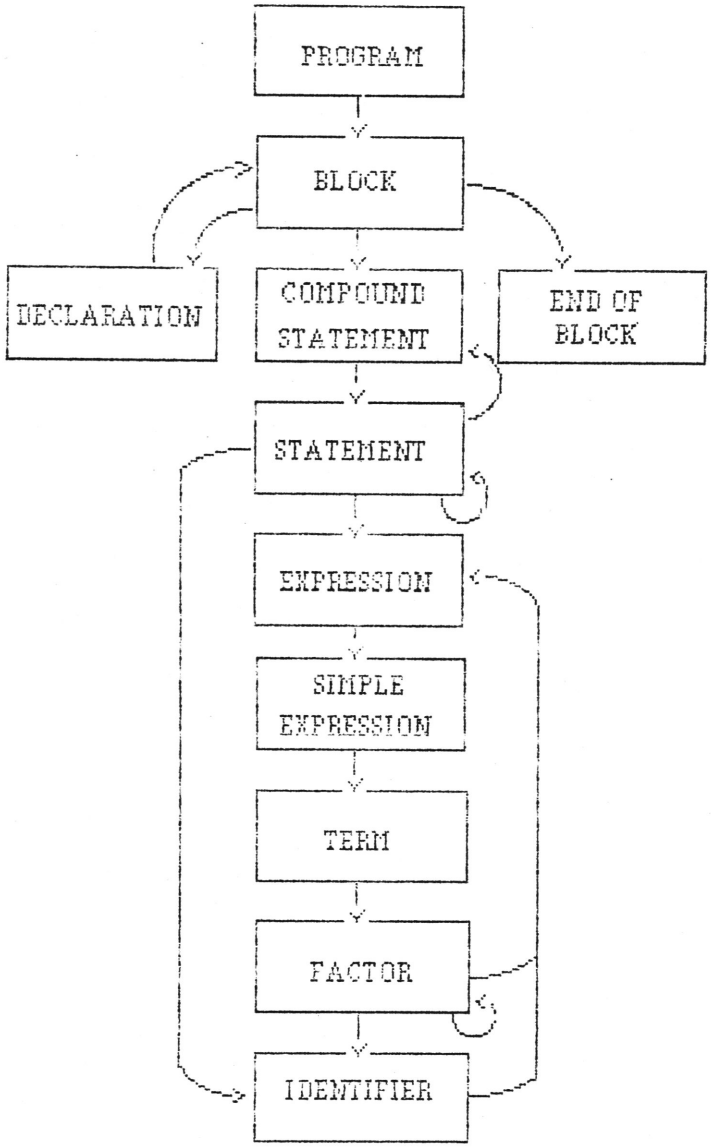

Fig. 5.1 Dependence diagram of the procedures in the translator

4.3 Tree-walking Technique of Translation

Before we proceed with the discussion of how the technique works, we shall first consider some of the terms that will be used in the discussion. We shall refer to a subset of the program as a fragment of a program (sometimes referred to as code fragment). In a PASCAL source program, it may be a declaration, a statement, an expression, or any other subset comprising the program. Also, we shall refer to a subset of a fragment as a

sub-fragment. Note that a fragment in some cases may also be a sub-fragment of another fragment of the program. For example, an expression is considered a sub-fragment of a statement and in some cases we also consider it a fragment in itself. Lastly, the term undecomposable component refers to parts of the program that can not be divided into sub-fragments and which require the translator to generate an equivalent object code, e.g., the variable 'x', the operator '+', etc.

The technique will answer the question of how to generate the code and not the question of what code will be generated for a particular program fragment. The latter question will be answered in later chapters.

The basic principle behind the tree-walking technique is to have a correct translation of the most basic or undecomposable component of a source code fragment. Since a fragment of a program is simply composed of sub-fragments which eventually are composed of undecomposable components, translating a fragment then is achieved by simply translating its basic components. This would consequently require the translator to be able to decide whether it is dealing with a decomposable or undecomposable fragment of a program. However, such will not be a problem because by examining the tokens returned by the scanner, a decision can certainly be made.

To illustrate the technique, we consider an example and see how translation can go on. Take for example, the statement

```
x := x + 1;.
```

We shall refer to the left component of the assigment as the destination and the right component as the source of the assignment. The analyser,

which is based on a bottom-up parsing principle, will recognize the statement as an assignment statement after scanning the assignment operator (':='). Since the fragment is identified as an assignment statement, the translator knows that it has to translate the destination of the assignment. It does the translation by generating the equivalent object code of that destination, i.e., 'x', and proceeds with the syntactic analysis of the source of the assignment. Further, the procedure that handles the translation of an assignment statement knows that the source of the assignment can take only one form and that form is an expression. So knowing the form, it invokes the procedures that handles expressions. Now that the control is in the procedure handling expressions, it has to decide what kind of expression it is. The scanner, again, will give the answer after returning the token for '+'. And since the operands of addition are undecomposable, equivalent object code will be generated for the expression. The delimeter ';' will signal the end of the expression and at the same time the end of the assignment statement. Thus the translator is again ready to translate another fragment of the program. The cycle is repeated until it encounters the token that signals the end of the program.

4.4 Reverse Polish Method of Evaluating Expressions

An expression is probably the most used PASCAL source code fragment. It is a sub-fragment of almost all PASCAL statements. As such, we discuss the method employed by the translator in translating PASCAL expressions.

The method, Reverse Polish, converts the expression in postfix form (also called Reverse Polish form). In a postfix form of an expression, the constituents of the expression, i.e., variables, constants, etc., are in the same order as the original expression but the operators are re-ordered

to comply with the operator precedence rules of the language. The best way of visualizing this rearrangement is to see examples of expressions and their corresponding postfix form. Below are some of these examples

Expression	Postfix Form
a - b + c * d / e	a, b, -, c, d, *, e, /, +
(a + b) * (c / d)	a, b, +, c, d, /, *
a * b > c - d	a, b, *, c, d, -, >

There are several ways of converting expressions to their postfix form and generating object code from them. One way is to build a tree with the nodes consisting of the constituents and operators of the expression, and touring the said tree in postorder. The second way is to store the constituents and operators of the expression in a stack (in postfix form, of course) and pop them one at a time generating object code each time. Another way is the method used by the translator described earlier. The method does not store the constituents and operators of the expression in a data structure like a tree or a stack. Instead the translator generates object code whenever a constituent of the expression is returned by the scanner. But if what is returned by the scanner is an operator, it decides whether to generate object code for the operator or remember the operator (because it has lower precedence) for later code generation.

CHAPTER 6

TRANSLATING PASCAL DECLARATIONS AND VARIABLES

The PASCAL declaration is divided into 5 parts: the label declaration, the constant definition, the type definition, the variable declaration, and the procedure (and function) declaration. The first three parts do not require object code to be generated but rather they are a source of information which will be useful in generating object code for the succeeding part of the program. They will be discussed in more detail in Chapter 10. The translation of the last two parts of the declaration, the variable declaration and procedure declaration, is primarily the subject of discussion of this chapter.

A variable in PASCAL may be in the form of an entire variable, a component variable, or a referenced variable. Indirectly, the translations of these variables will be discussed together with the way they are declared. Variables could also be classified in terms of where in the program access to storage allocated to them is made. This clasification is local and non-local variables. Translation of local and non-local variables will be discussed in the last two sections of this chapter.

6.1 Translating Variable Declarations

The variable declaration part of a PASCAL program is where association of an identifier and/or a data type with a new variable is made. The data type of a variable determines the set of values that can be assigned to it at run-time. On declaration of a new variable, it is

necessary that enough space be allocated in the stack to hold the value that will be assigned to the variable. How much storage it needs will depend on the type of the variable and how it will be accessed in the program.

In the succeeding sections, we shall be concerned with translation of the declaration and access to variables of the following types: standard type, scalar type, structured type (particularly, array type) and record type.

6.1.1 Translating Standard Type Variables

The standard types are those types provided by most computer systems. They include whole numbers, logical truth values, set of printable characters, and in some large systems floating point numbers. PASCAL provides four standard types. These types are the integer type, boolean type, character type and real type.

The integer type comprises the subset of whole numbers whose range is implementation dependent. The smallest storage that can be allocated in the stack for an integer type variable is a SLIM cell. This suggests that the equivalent instruction of those variables declared with integer type is the SLIM's M1 instruction.

The boolean type can assume only the values true or false. The PASCAL standard states that a true value should be internally represented by the value 1 and the boolean value false represented by the value 0. But these internal representation of boolean values will only become important when comparing two boolean values, i.e., a true value is greater than a

false value. On the other hand, SLIM's internal representation of false is 0 and true is -1. A boolean value of 1 in SLIM is undefined. So to avoid further problems, we represent a true value by -1 and just negate it when doing the comparison between boolean values. This incompatibility between PASCAL and SLIM will be resolved in the section on translation of boolean expressions.

Although the possible boolean values will actually fit into one byte of storage, for faster access and uniformity of variable access instructions, a cell of storage is used to allocate stack space to a variable of type boolean. Hence, the translation of a boolean type variable declaration is M1. Of course, this decision is not sound when implementing PASCAL in a computer with a limited memory. But for a translator whose aim is to cut as much as possible the time required to compile source programs, the translation of boolean type variables might as well be the same as the other standard type variables to avoid a test being carried out at compile time before generating object code for standard type variable declarations.

The character type comprises the set of all printable characters. The set of characters recognized by the machine and the internal bit-strings used to represent the characters are machine-dependent. But most machines use what is now becoming the standard character set and bit-string encoding, the American Standard for Information Interchange (ASCII) code.

Variables of character type are similar to boolean type, i.e., their internal representation fits into one byte of storage. But for reasons similar to why a cell of storage was used to allocate a variable of boolean type, character type identifiers are allocated a cell of storage.

Similarly, a stack space is allocated to a character type variable by the SLIM instruction M1.

The last standard type variable is the real type. It denotes a subset of real numbers, specifically it comprises the floating point number of the machine. This type is also implementation-dependent like the integer type, but unlike the integer type it usually requires at least 32 bits of storage. This makes the translation of real type variables dependent on the SLIM cell size. If the SLIM cell size is 16 bits, allocation of storage for the variable will require the instruction M2 and if its 32 bits or more the M1 instruction will do. In the succeeding discussions, it is assumed that the cell size is 32 bits; thus a real type variable will require a cell of storage.

To summarize the SLIM code necessary to allocate storage spaces to standard types, we take an example declaration and write opposite it the SLIM translation.

Example:

PASCAL		SLIM
var count:	integer;	M1
found:	boolean;	M1
letter:	char;	M1
root:	real;	M1 or M2 (if cell size < 32)

The initial values of these variables is whatever is stored in the cell allocated to it when the declaration is made. This is satisfactory since the PASCAL standard states that the initial value of a variable is undefined until an assignment statement which assigns a value to it is

executed. But some implementations override this specification of the language and set the initial values of variables to a certain value. This extension will require more than one instruction. Note that SLIM allows only transfer of data from the accumulator to memory, and vice versa. To implement the extension then will mean generating the L and P instructions. Using the same example as above, the translations with the extension will then be

PASCAL	SLIM	Initial Value
var count: integer;	L0 P	Zero
found: boolean;	L0 P	False
letter: char;	L'*S P	Space character
root: real;	L0.0 P	Zero.

Access to standard type variables can well be illustrated by considering examples of PASCAL statements and their SLIM equivalents. Below is a summary of access instructions for the standard type variables:

PASCAL	SLIM	Type
count := 100	L100 S(count)	Integer
found := true	L-1 S(found)	Boolean
letter := 'C'	L'C S(letter)	Char
root := 1.25	L1.25 S(root)	Real

The operand of the store instruction is equal to the address of the cell reserved for the variable during its declaration. Of course, the computation of the operand depends on what kind of identifier it is, i.e., local, non-local, etc.

6.1.2 Translating Scalar Type Variables

A scalar type by definition is an ordered set of values by enumeration of the identifiers which denote these values. The declaration of a scalar type variable introduces not only a new variable, but at the same time a new set of constant identifiers. For example, a declaration

```
number: (zero, one, two);
```

introduces the scalar type variable 'number' and at the same time the set of constant identifiers, 'zero', 'one', 'two' whose integer values are 0, 1, and 2 respectively. These values are assigned according to their position in the enumeration. It is then obvious that scalar type variables can assume only whole number values and would require one cell of storage. The SLIM M1 instruction therefore is the translation of the declaration of a scalar type variable.

Access to scalar type variables then, is no different from access to standard type identifiers. To illustrate this, we consider the declaration

```
day: (mon, tue, wed, thur);,
```

access to the variable 'x' can take any one of the following forms:

PASCAL	SLIM
day := mon;	L0 S(day)
day := tue;	L1 S(day)
day := wed;	L2 S(day)
day := thur;	L3 S(day)

where 'day' in the store instruction is the address of the cell allocated to variable 'day'.

6.1.3 Translating Array Type Variables

An array is a structure consisting of a fixed number of components with each component having the same type, i.e., array of integer, array of record, array of array, etc. An array is a random-accessed storage area indexed according to the value of the subscript. Access to an element of the array is an expensive business. It involves the calculation of the address of the element to be accessed. It is this expensive nature which lead to the development of several methods of accessing an element of an array. Two commonly used methods are the multiplicative subscript calculation and the indirect access via pre-calculated vectors of addresses. The translation of an array declaration will depend entirely on which method is used.

The two methods of translating an array declaration and access to an element of an array will be discussed in the next two sections. The suitablity of each method for translating PASCAL arrays will also be discussed.

6.1.3.1 Indirect Access Via Pre-calculated Addresses Method

We shall discuss indirect access via pre-calculated addresses method of accessing an element of an array by first considering how translation is carried out for a two-dimensional array and generalizing it to an n-dimensional array (with n greater than or equal to 1).

The setting up of a two-dimensional array ($m_1 x m_2$) is done by allocating enough storage for the vectors pointing to the rows of the array

LE(b+m_1+1) P M(m_1+1),

where b is the current stack offset, and then enough storage for the elements of the array

LE(b+(m_1+1)+(m_1*m_2+1)) P M(m_1*m_2+1).

Letting

$a = b + m_1$

and

$v_1 = b + (m_1 + 1) + (m_1 * m_2 + 1)$,

the vector of addresses which points to each row of the array is initialized by

for := 0 to m_1 - 1 do a[i] := $v_1 + m_2 * i$.

This setting up of the array should be done during array declaration.

After setting up the array correctly, the translation of

```
a[i, j] := w
```

is

SLIM	Comment
La P Li PLH L!H	Load address of the row where a[i,j] is
P Lj PLH L!H	Load address of a[i,j]
P	Push it onto the stack
Lw	Load w
SH	Store w in a[i,j]

and the translation of

```
w := a[i, j]
```

is

SLIM	Comment
La P Li PLH L!H	Load address of the row where a[i,j] is
P Lj PLH L!H	Load address of a[i,j]
L!O	Load value of a[i,j]
Sw	Store it in w.

Now, consider an n-dimensional array ($m_1 x m_2 x m_3 x \ldots x m_n$) where $m_i = hi_i - lo_i + 1$, hi_i and lo_i are the upper bound and lower bound respectively of the i'th dimension, b as the current stack offset, the element size of the array equal to one, and assuming a backward growing stack. The SLIM code necessary to allocate space to the elements of the array and the vectors of addressess are the following:

Space for the vector of addresses

$LE(b+m_1+1)\ P\ M(m_1+1)$ [1]

$LE(b+(m_1+1)+(m_1*m_2+1))\ P\ M(m_1*m_2+1)$ [2]

$LE(b+(m_1+1)+(m_1*m_2+1)+(m_1*m_2*m_3+1))\ P\ M(m_1*m_2*m_3+1)$ [3]

...

...

...

$LE(b+(m_1+1)+(m_1*m_2+1)+(m_1*m_2*m_3+1)+ \ldots +(m_1*m_2*m_3* \ldots *m_{n-1}))$

$P\ M(m_1*m_2*m_3* \ldots *m_{n-1}+1)$ [n-1]

Space for the n-dimensional array

$LE(b+(m_1+1)+(m_1*m_2+1)+(m_1*m_2*m_3+1)+ \ldots +(m_1*m_2*m_3* \ldots *m_n))$

$P\ M(m_1*m_2*m_3* \ldots *m_{n+1})$ [n].

Which means that the total number of cells necessary to allocate storage to an n-dimensional array (with element size equal to one) is

$$m_1+(m_1*m_2)+(m_1*m_2*m_3)+ \ldots +(m_1*m_2*m_3* \ldots *m_n) + n.$$

After allocating enough storage to the array and to the vectors of addresses necessary to access this array, these vectors have to be initialized. By letting

$a := b+m_1+1;$

$v_1 := b+m_1+(m_1*m_2)+2;$

$v_2 := b+m_1+(m_1*m_2)+(m_1*m_2*m_3)+3;$

...

...

...

$$v_{n-1} := b+m_1+(m_1*m_2)+(m_1*m_2*m_3)+ \ldots +(m_1*m_2*m_3* \ldots *m_n)+n;$$

the pointers are initialized by the following:

for i:= 0 to m_1-1 do a[i] := v_1+m_2*i

for i:= 0 to m_1*m_2-1 do v_1[i] := v_2+m_3*i

for i:= 0 to $m_1*m_2*m_3-1$ do v_2[i] := v_3+m_4*i

...

...

...

for i:= 0 to $m_1*m_2*m_3* \ldots *m_{n-1}-1$ do v_{n-2}[i] := $v_{n-1}+m_n*i$

which can be translated to SLIM and is described in Chapter 8. Figure 6.1 shows part of the stack after allocating storage and initializing the vectors of addresses.

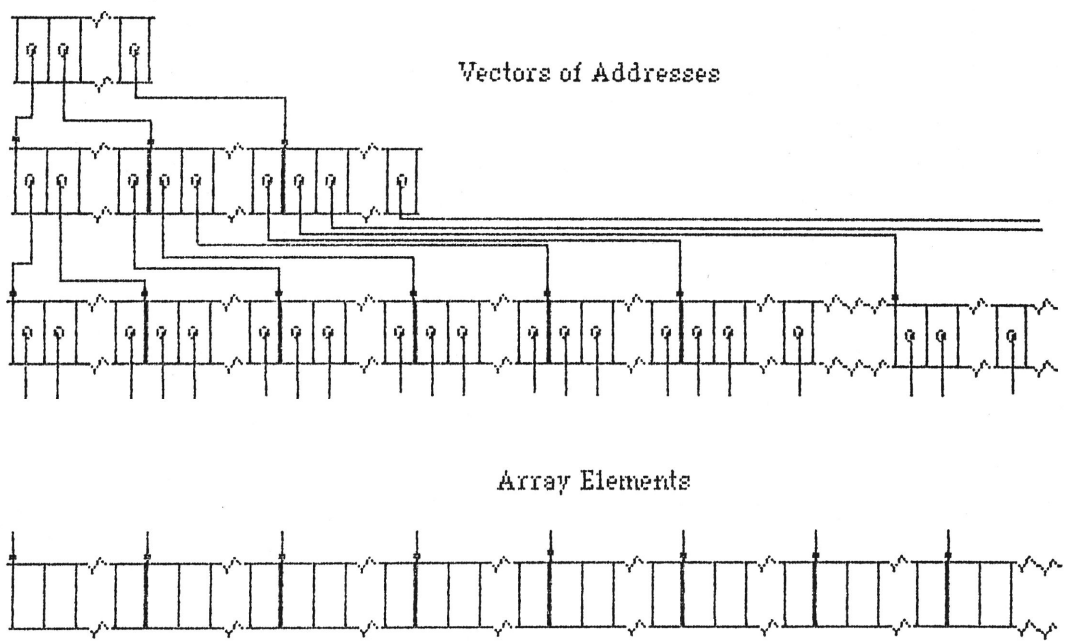

Fig. 6.1 The stack after the declaration of an n-dimensional array. The access method used is the indirect access via pre-calculated addresses method.

After setting up the array correctly, the SLIM code for accessing an element of the array will then be

PASCAL	SLIM	Comment
$a[i_1,i_2,i_3,\ldots,i_n] := w$	La P Li_1 PLH L!H	Load $a[i_1]$
	P Li_2 PLH L!H	Load $v_1[i_2]$
	P Li_3 PLH L!H	Load $v_2[i_3]$
	...	
	...	
	...	
	P Li_n PLH L!H	Load address of $a[i_1,i_2,i_3,\ldots,i_n]$
	P	Push it into the stack
	Lw	Load w
	SH	Store w in $a[i_1,i_2,i_3,..,i_n]$
$w := a[i_1,i_2,i_3,\ldots,i_n]$	La P Li_1 PLH L!H	Load $a[i_1]$
	P Li_2 PLH L!H	Load $v_1[i_2]$
	P Li_3 PLH L!H	Load $v_2[i_3]$
	...	
	...	
	...	
	P Li_n PLH L!H	Load address of $a[i_1,i_2,i_3,\ldots,i_n]$
	L!0	Load content of $a[i_1,i_2,i_3,\ldots,i_n]$
	Sw	Store $a[i_1,i_2,i_3,\ldots,i_n]$ in w.

We should note that array indices in PASCAL can be in the form of an arithmetic expression. Which is why, with the translator described

(one-pass), the instructions PLH L!H are necessary and the set of instructions Li PLH L!H can not be represented by an equivalent instruction Lli.

Indirect access using the pre-calculated addresses method of array access is known for its speed in accessing an element of an array. But in exchange for the speed, it requires an amount of memory that increases significantly with the dimension of the array. Aside from memory usage, one should also take careful consideration of the overhead introduced by the initialization process. The initialization process itself involves one-dimensional array access. Certainly, this method is ideal if your machine has enough memory to hold the vectors of addresses.

One possible inconvenience in using this method for PASCAL array access is when an array is passed by value to a procedure. We should note that when an array is passed by value, the whole array including its vectors of addresses have to be copied to the called procedure. But, since the addresses computed during the initialization of the array are applicable only to the procedure where it is declared, the initialization procedures have to be redone in the called procedure.

6.1.3.2 Multiplicative Subscript Calculation Method

The multiplicative subscript calculation method of accessing an element of an array was first used in FORTRAN. The method works by transforming an $m_1 x m_2 x m_3 x \ldots x m_n$ array, say a, into an equivalent one-dimensional array with $m_1 x m_2 x m_3 x \ldots x m_n$ elements. Instead of using n

indices to access an element of the array, the n indices are used to calculate the one index (call it newindex) that tells the relative position of the element to be accessed in the equivalent one-dimensional array. The transformation can be carried out by simplifying the equation

$$\begin{aligned}\text{address of } a[i_1,i_2,i_3,\ldots,i_n] = \text{address of } a[lo_1,lo_2,lo_3,\ldots,lo_n] \\ + (i_1-lo_1)*m_2*m_3*\ \ldots\ *m_n \\ \\ + (i_2-lo_2)*m_3*m_4*\ \ldots\ *m_n \\ + (i_3-lo_3)*m_4*m_5*\ \ldots\ *m_n \\ \ldots \\ \ldots \\ \ldots \\ + (i_{n-1}-lo_{n-1})*m_n \\ + (i_n-lo_n)\end{aligned}$$

to

$$\begin{aligned}&\text{address of } a[i_1,i_2,i_3,\ldots,i_n] \\ &\quad = \text{address of } a[0,0,0,\ldots,0] \\ &\quad + (\ \ldots\ (((((i_1*m_2)+i_2)*m_3)+i_3)*m_4)+i_4\ \ldots\ldots\ *m_n)+i_n\end{aligned}$$

where lo_i and hi_i are the lower bound and upper bound respectively of the i'th dimension, $m_i=hi_i-lo_i+1$ is the size of the i'th dimension. A requirement in using this transformed formula is to know the values of the terms of the formula at run-time, i.e., address of a[0 0,0,...,0] and n-1 values of m_i. It does not really matter whether a[0,0,0,...,0] falls outside the bounds of the array since the formula is interested only in its address and not its content.

To take advantage of this transformation, there should be a way of knowing, at run-time, the lower and upper bounds of the array and the address of a[0,0,0,...,0]. These values are known at compile time, although some of them are not known directly but they can be derived by a computation. One way of making these values available at run-time is to store the information required right before or after the array. This storage for information about the array is what is commonly known as the dope vector. With the dope vector, part of the stack after an array declaration is shown in figure 6.2.

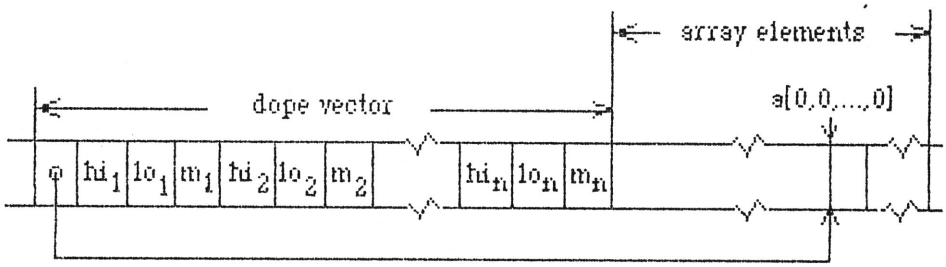

Fig. 6.2 The stack after the declaration of an n-dimensional array. The access method used is the multiplicative subscript calculation method.

Since the transformed formula consists only of the address of a[0,0,0,...,0] and the length of each dimension, and of course the indices, one may wonder why the dope vector contains the lower and upper bounds. The usefulness of storing the bounds will become obvious in the section on subscript checking.

PASCAL allows the passing of an array by value to a procedure. It should therefore be arranged that the information in the dope vector be position independent, i.e., wherever the array is copied in the stack the dope vector should be able to provide the correct information for accessing an element of the array. This could easily be arranged by changing the

entry on the address of a[0,0,0,...,0] for it is only the position dependent information in the dope vector we considered so far. Note that the address of a[0,0,0,...,0] is true only to the procedure where it is declared. To make sure that the information in the dope vector is position independent, the address of a[0,0,0,...,0] can be replaced by its offset from the first cell of the storage allocated to the array. Since the size of arrays in PASCAL is static, this information will always hold true wherever the array is copied. The address of the zeroth element then can be calculated by adding this offset to the address of the said first cell.

First, let us consider the case where an array has two subscripts, i.e., a two-dimensional array (m_1xm_2). The declaration of the array will require the following code to be generated:

L(7+m_1*m_2)
PLhi$_1$ PLlo$_1$ PLm$_1$
PLhi$_2$ PLlo$_2$ PLm$_2$
P M(m_1*m_2)

After allocating enough storage to the array, assuming a backward growing stack, the translation of

a[i, j] := w

is

SLIM	Comment
La SE1	Load address of the first cell allocated to the array and store it in E1

P L!0	Load offset of a[0,0] (note that the offset is in the said first cell)
+H P	Compute address of a[0,0] and push it into the stack
L0	Clear the accumulator in preparation for the computation of $(i*m_2)+j$
P Li +H	Load i
P LIE1 L!-6	Load m_2
*H	Compute $i*m_2$
P Lj	Load j
+H	Compute i*m2+j
+H	Add $(i*m_2+j)$ to address of a[0,0]
P	Push it onto the accumulator
Lw	Load w
SH	Store w in a[i,j]

and the translation of

```
w := a[i,j]
```

is

SLIM	Comment
La SE1	Load address of the first cell allocated to the array and store it in E1
P L!0	Load offset of a[0,0] (note that the offset is in the said first cell)
+H P	Compute address of a[0,0] and push it into the stack
L0	Clear the accumulator in preparation for the computation of $(i*m_2)+j$
P Li +H	Load i
P LIE1 L!-6	Load m_2
*H	Compute $i*m_2$

```
P Lj                    Load j
+H                      Compute i*m2+j
+H                      Add (i*m2+j) to address of a[0,0]
L!0                     Load content of a[i,j]
Sw                      Store a[i,j] in w.
```

Now, consider an n-dimensional array ($m_1 x m_2 x m_3 \ldots x m_n$) where $m_i = hi_i - lo_i + 1$, hi_i and lo_i are the upper and lower bounds respectively of the i'th dimension, b is the current stack offset, s_dope=(3*n)+1 is the number of cells occupied by the dope vector, e_size is the number of fields in each element of the array, and a backward growing stack is assumed. Allocation of storage to an array then would mean generating the following SLIM code:

```
L(s_dope+m1*m2*m3* ... *mn)
PLhi1 PLlo1 PLm1
PLhi2 PLlo2 PLm2
PLhi3 PLlo3 PLm3
  ...
  ...
  ...
PLhin PLlon PLmn P
M((m1*m2*m3* ... *mn)*e_size)
```

It is then obvious that the number of cells required for array access in this method is

```
(3*n)+1+((m1*m2*m3* ... *mn)*e_size)
```

which is much less than what is needed by the indirect access via

pre-calculated addresses method. Further, no initialization of vectors takes place in this method.

However, in consequence of the smaller storage requirement, access to an element of an array requires extensive computation of the address of the element at run-time. This can be shown by the SLIM code necessary to access an element of an array using the multiplicative address calculation (see below).

PASCAL

$a[i_1, i_2, i_3, \ldots, i_n] := w$

SLIM	Comment
La SE1	Load address of the first cell allocated to the array and store it somewhere, in this case E1
P LI0	Load offset of a[0,0,0,...,0] from the said first cell
+H P	Compute the address of a[0,0,0,...,0] and push it into the stack
L0	Clear the accumulator in preparation for the computation of $(\ldots(((i_1*m_2)+i_2)*m_3)+i_3\ldots*m_n)+i_n$
P Li$_1$ +H	i_1
P LIE1 LI-6	m_2
*H	i_1*m_2
P Li$_2$ +H	$(i_1*m_2)+i_2$
P LIE1 LI-9	m_3
*H	$((i_1*m_2)+i_2)*m_3$
...	

...

...

P Li_n +H	$(...(((i_1*m_2)+i_2)*m_3)+i_3 ...*m_n)+i_n$ [*]
+H	a[0,0,...,0] + [*] = address of $a[i_1, i_2, ..., i_n]$
P	Push result into the stack
Lw	Load w
SH	Store w in $a[i_1, i_2, i_3, ..., i_n]$

PASCAL

$w := a[i_1, i_2, i_3, ..., i_n]$

SLIM	Comment
La SE1	Load address of the first cell allocated to the array and store it somewhere, in this case E1
P LI0	Load offset of a[0,0,0,...,0] from the said first cell
+H P	Compute the address of a[0,0,0,...,0] and push it into the stack
L0	Clear the accumulator in preparation for the computation of $(...(((i_1*m_2)+i_2)*m_3)+i_3 ...*m_n)+i_n$
P Li_1 +H	i_1
P LIE1 LI-6	m_2
*H	i_1*m_2
P Li_2 +H	$(i_1*m_2)+i_2$
P LIE1 LI-9	m_3
*H	$((i_1*m_2)+i_2)*m_3$

...

...

...

P Li_n +H	$(\ldots(((i_1*m_2)+i_2)*m_3)+i_3 \ldots*m_n) + i_n$ [*]
+H	a[0,0,...,0] + [*] = address of $a[i_1, i_2, \ldots, i_n]$
LIO	Load content of $a[i_1, i_2, i_3, \ldots, i_n]$
Sw	Store it in w.

The cell having the address E1 is a general purpose storage used for temporary storage of values. It was introduced primarily to facilitate translation involving arrays. Its introduction will become more important later, when subscript checking is introduced into the translation of an array.

6.1.3.3 Subscript Checking

The values of the subscripts of an array are usually known at run-time. Moreover, they can assume values that lie outside the allowed range. When this happens, the execution of the program is unpredictable. It may access wrong values, or at worst, it may try to write to a part of memory not allocated to it. To avoid this sort of problem, subscript checking may be implemented.

Subscript checking may be carried out at run-time, although to some extent it can be done at compile time, e.g., when the subscripts are constants. The check can be carried out by comparing the subscripts with the bounds specified during declaration. If the subscripts lie outside the range, the execution of the program should be aborted.

Again, since most of the checks must be done at run-time, arrangements must be made to make the bounds accessible at run-time. In the case of an implementation where multiplicative subscript calculation is

used for accessing an array, checks can be carried out immediately because array bounds are available in the dope vector. But in the case where indirect access via pre-calculated addresses method is used, extra instructions must be added to the existing instructions necessary to allocate storage to arrays. To be able to do subscript checks, the declaration of an array using indirect method via pre-calculated addresses method is extended to the following:

Space for the vector of addresses

LE(b+m_1+1) P M(m_1+1) [1]

LE(b+(m_1+1)+(m_1*m_2+1)) P M(m_1*m_2+1) [2]

LE(b+(m_1+1)+(m_1*m_2+1)+(m_1*m_2*m_3+1)) P M(m_1*m_2*m_3+1) [3]

...

...

...

LE(b+(m_1+1)+(m_1*m_2+1)+(m_1*m_2*m_3+1)+ ... +(m_1*m_2*m_3* ... *m_{n-1}))

P M(m_1*m_2*m_3* ... *m_{n-1}+1) [n-1]

Space for the n-dimensional array

LE(b+(m_1+1)+(m_1*m_2+1)+(m_1*m_2*m_3+1)+ ... +(m_1*m_2*m_3* ... *m_n))

P M(m_1*m_2*m_3* ... *m_{n+1}) [n]

Space for the dope vector

Lhi_1 $PLlo_1$

$PLhi_2$ $PLlo_2$

$PLhi_3$ $PLlo_3$

...

...

...

$PLhi_n$ $PLlo_n$ P

Subscript checking can be implemented by generating the SLIM code

SLIM	Comment
P P	Make two copies of the index
PLhi <H	Is the upper bound < the index?
F@1	
@2: L2 CIO87 D1 Q	Yes! Send error message and quit
@1: Llo >H	No! Is the lower bound > index?
F@3	
J@2	Yes! Send error message and quit
@3: LH	No! Restore value of the accumulator

immediately after the translation of an index of an array.

The translation of an array with subscript checking for the two methods can be summarized as follows

Indirect Access Via Pre-calculated Address Method

PASCAL

$a[i_1, i_2, i_3, \ldots, i_n] := w$

SLIM	Comment
La P Li_1	
	Start of subscript check
P P PLv_n LI-1 <H	Is $hi_1 < i_1$?
F@1	
@2: L2 CIO87 D1 Q	Yes! Send error message and quit
@1: Lv_n LI-2 >H	Is $lo_1 > i_1$?
F@3	
J@2	Yes! Send error message and quit
@3: LH	No! Restore value of accumulator
	End of subscript check

```
PLH LIH                    Load a[i1]
P Li2
                           Subscript check of i2
P P PLvn LI-3 <H
F@4
@5: L2 CIG87 D1 Q
@4: Lvn LI-4 >H
F@6
J@5
@6: LH                     End of subscript check
PLH LIH                    Load v1[i2]
P Li3
                           Subscript check of i3
P P PLvn LI-5 <H
F@7
@8: L2 CIG87 D1 Q
@7: Lvn LI-6 >H
F@9
J@8
@9: LH                     End of subscript check
PLH LIH                    Load v2[i3]
...
...
...
P Lin
                           Subscript check of in
P P PLvn LI-(2*n-1)
<H
F@(3*(n-1)+1)
@(3*(n-1)+2):
L2 CIG87 D1 Q
@(3*(n-1)+1):
Lvn LI-(2*n) >H
F@(3*(n-1)+3)
J@(3*(n-1)+2)
```

@(3*(n-1)+3): LH	End of subscript check
PLH LIH	Load address of $a[i_1, i_2, i_3, \ldots i_n]$
P	Push it into the stack
Lw	Load w
SH	Store w in $a[i_1, i_2, i_3, .., i_n]$

Multiplicative Subscript Calculation Method

PASCAL

$w := a[i_1, i_2, i_3, \ldots, i_n]$

SLIM	Comment
La SE1	Load address of the first cell allocated to the array, in this case E1
P LI0	Load offset of a[0,0,0,...,0] from the said first cell
+H P	Compute the address of a[0,0,0,...,0] and push it into the stack
L0	Clear the accumulator in preparation for the computation of $(\ldots(((i_1*m_2)+i_2)*m_3)+i_3\ldots*m_n)+i_n$
P LI_1	Start of subscript check
P P PLIE1 LI-1 <H	Is $hi_1 < i_1$?
F@1	
@2: L2 CIG87 D1 Q	Yes! Send error message and quit
@1: LIE1 LI-2 >H	Is $lo_1 > i_1$?
F@3	
J@2	Yes! Send error message and quit
@3: LH	Load value of i_1
	End of subscript check
+H	i_1

```
P LIE1 LI-6          m2
*H                   i1*m2
P Li2
                     Subscript check of i2
P P PLIE LI-4 <H
F@4
@5: L2 CIG87 D1 Q
@4: LIE LI-5 >H
F@6
J@5
@6: LH               End of subscript check
+H                   (i1*m2)+i2
P LIE1 LI-9          m3
*H                   ((i1*m2)+i2)*m3
...
...
...
P Lin
                     Subscript check of  in
P P PLIE1 LI-(3*(n-1)+1)
<H
F@ (3*(n-1)+1)
@(3*(n-1)+2):
 L2 CIG87 D1 Q
@(3*(n-1)+1):
 LIE1 LI-(3*(n-1)+2) >H
F@(3*(n-1)+3)
J@(3*(n-1)+2)
@(3*(n-1)+3): LH     End of subscript check
+H                   (...(((i1*m2)+i2)*m3)+i3 ...*mn)+in [*]
+H                   Compute address of a[i1,i2,...,in]
P                    Push it into the stack
Lw                   Load w
SH                   Store w in a[i1,i2,i3,...,in]
```

An alternative way of implementing subscript checking in SLIM is to write a procedure which will do the check. This procedure can then be incorporated in the SLIM library routines.

To be more specific, the procedure we mentioned is a procedure with two parameters. The first parameter is the index of the array and the second is the address of the first cell allocated to the array. The address of the first cell allocated to the array is enough to make the bounds of the array accessible to the procedure.

In this method, the amount of code necessary to be generated to translate an array with subscript checking can then be reduced significantly. But this alternative method involves the instructions C (procedure call) and R (procedure return) which are simulated by several lines of target machine's assembler code. Thus the check might be slower compared to the original check we mentioned. To illustrate the reduction, instead of the translation

```
P P PLIE1 L!-1 <H
F@1
@2: L2 CIG87 D1 Q
@1: LIE1 L!-2 >H
F@3
J@2
@3: LH
```

we have the equivalent translation using the descibed procedure as

```
P               Make a copy of the index
PLIE1 CIGk D2   Pass the parameters and call the procedure
LH              Restore the value of the accumulator.
```

The call to a procedure from the SLIM library routines, e.g., CIGk D2, will be discussed in more detail in Chapter 8.

Although subscript checking will almost double the object code necessary to translate a source program fragment involving array access, most implementations incorporate the check for it can save the programmer many hours of frustration.

After this section, translation involving arrays will usually be presented using only one method, the multiplicative subscript calculation method and without subscript checking.

6.1.4 Translating Record Type Variables

A record structure, like an array, is a structure with a fixed number of components. But the components, unlike an array, can be of different types. Indices in records, called record offsets, are of fixed size.

As for any other identifier type, identifiers declared to be of record type require storage space to be allocated. To set up a record is no different from the set up procedures of the types that compose the record itself. In fact, when an identifier is declared with type record, the translator will examine the components of the record and call the routines necessary to allocate space to the component. The translator does not have any special routine for translating the declaration of a record type variable. To illustrate this point, consider the following declaration

```
type recd1 = record
          a1: integer;
          b1: array[1..5] of char;
     end;
     recd2 = record
          a2: char;
          b2: array[-1..1] of real;
          c2: recd1;
     end;
     recd3 = record
          a3: recd2;
          b3: (one, two, three);
     end;

var id: recd3;
```

space allocation to this declaration, i.e.,

```
var id : recd3;
```

can proceed as for the declaration

```
var:  a2: char;
      b2: array[-1..1] of real;
      a1: integer;
      b1: array[1..5] of char;
      b3: (one, two, three);.
```

But, of course, when it comes to accessing this storage space, the two declarations are different. Figure 6.3 shows a snapshot of the stack after the declaration of variable 'id' as recd3, i.e., var id: recd3. The figure assumes a forward growing stack.

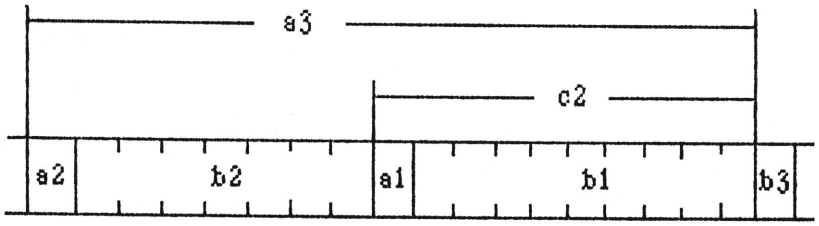

Fig. 6.3 The stack after declaring a variable as recd3.

Accessing an element of a record is more efficient than accessing an element of an array. This effieciency can be ascribed to the fact that record offsets are known at compile time while array subscripts are mostly known at run-time. The table below shows some examples of translations involving record access using the declaration given above. Before the example, a summary of record offsets is presented to help explain, indirectly of course, some of the translations.

Declaration	Record Offset
type recd1 = record	
a1: integer;	0
b1: array[1..5] of char;	1
end;	
recd2 = record	
a2: char;	0
b2: array[-1..1] of real;	1
c2: recd1;	6
end;	
recd3 = record	
a3: recd2;	0
b3: (one, two, three);	18
end;	

Examples:

PASCAL	SLIM
w := id.a3.a2	Lid L!0 Sw
id.a3.a2 := w	Lid P Lw SH
w := id.b3	Lid -18 L!0 Sw
id.b3 := w	Lid -18 P Lw SH
w := id.b2[1]	Lid -1 P SE1 L!0 +H P L0 P L1 +H +H L!0 Sw
id.b2[1] := w	Lid -1 P SE1 L!0 +H P L0 P L1 +H +H P Lw SH
w := id.c2.a1	Lid -6 L!0 Sw
id.c2.a1 := w	Lid -6 P Lw SH
w := id.c2.b1[5]	Lid -6 -1 P SE1 L!0 +H P L0 P L5 +H +H L!0 Sw
id.c2.b1[5] := w	Lid -6 -1 P SE1 L!0 +H P L0 P L5 +H +H P Lw SH.

Again, we stress the importance of the cell whose address is E1. It is in this type of translation where it is most useful, i.e., records having array components. Note that if you include subscript checking to the translation of 'w := id.c2.b1[5]', you will end up computing the address of 'id.c1.b1' which is 'Lid -6 -1' to access the bounds of the array. But since that address is stored in E1 after it was computed the first time, accessing the bounds can then be done by just loading that stored address (L!E1) and indexing to access the cells where the bounds are stored. To illustrate this, we rewrite the translation of 'w := id.c2.b1[5]', incorporating subscript checking, with and without the temporary storage, E1.

PASCAL	With E1	Without E1
w := id.c2.b1[5]	Lid -6 -1 P SE1 L!0 +H P L0 P L5	Lid -6 -1 P L!0 +H P L0 P L5

```
    /* Subscript Checks  Starts */
P P PLIE1 LI-1 <H        P P PLid -6 -1 LI-1 <H
F@1                      F@1
@2: L2 CIG87 D1 Q        @2: L2 CIG87 D1 Q
@1: LIE1 LI2 >H          @1: Lid -6 -1 LI-2 >H
F@3 J@2                  F@3 J@2
@3: LH                   @3: LH
       /* End of Checks */
+H +H LIO Sw             +H +H LIO Sw
```

The use of the temporary storage, E1, will even become more useful when the forms of the record type variable are more complicated. One example of such a form is

a.b[i*4+8].c[j+8].d[7].

Without the temporary storage (E1), translating this form of a record variable and incorporating subscript checking will entail much code to be remembered and later generated.

6.2 Translating Procedure Declarations

The procedure declaration part of a PASCAL program, like the program itself, is treated as a block. But unlike the program, it is a block with formal parameters appearing as variables local to it. It is composed of a program heading and a block. The block part in turn consists of the declaration and statement parts. The declaration part of a procedure is exactly the same as the program declaration discussed earlier. The statement part will be the subject of discussion of Chapter 8. In this section therefore, we shall be concerned with the translation of the procedure heading and other matters not covered by the declaration and statement parts.

The heading of the procedure is divided into three parts: procedure name, formal parameters, and result type. The result type being present only when the procedure returns a value, i.e., in PASCAL it is declared as a function. Below is a summary of the translation of each part of a procedure heading.

Part	SLIM	Comment
procedure name	$"name"	is not really necessary, but was incorporated to make the code readable
	@entry:	Entry point of the procedure
formal parameters	Dm	m is the number of formal formal parameters
	M1	Allocate space for the general purpose cell, E1, used for translating arrays
result type	M1	Allocate space for the function identifier, E2. Generated only when the procedure is declared as a function.

The computation of the number of formal parameters, m, is dependent on the amount of storage required to store the parameters. This amount of storage is known for each kind of parameter. Firstly, a parameter may be a value parameter. In which case the amount of storage required will depend on the type of the variable, i.e., standard type requires one cell of storage, array type's requirement depends on the method used for accessing its elements, etc. The second kind of parameter is the variable (var) parameter. It only requires one cell of storage; storage for the address of

the variable. Finally, procedure parameters which require two cells of storage; one for the entry point and another for the value of the static link.

When a procedure is declared as a function, it is assumed that the procedure returns a value. Further, the function identifier is treated as a variable local to the procedure and is where the value to be returned is stored. From the translation of the heading, we note that when a procedure is declared as a function, the second M1 instruction is generated. This M1 instruction allocates storage space for the function identifier. Note that in PASCAL assignment to a function identifier is legal, hence the storage space. The assignment to the function identifier can be done anywhere in the body of the procedure. To make sure that the value assigned to the function identifier is returned, the LIE2 instruction is generated before the end of the procedure.

Examples of translations for a procedure that returns a value (function) and one that does not return a value (procedure) are given below.

Procedure

PASCAL	SLIM
procedure name(	$"name"
	@1:
a: integer;	
var b:integer;	
procedure c);	D4 M1
begin	
{ procedure body }	
end;	

Function

PASCAL	SLIM
function name(	$"name"
	@1:
a: integer;	
var b:integer;	
procedure c	D4 M1
): real	M1
begin	
{ procedure body }	
name := 1.23	L1.23 SE2
{ procedure body }	
end;	LIE2

In the procedure that is declared as a function, note the instruction LIE2 at the end of the procedure body. Such an instruction ensures that the value is returned by the procedure.

After this section, the term procedure will usually refer to both procedures that return values and those that do not.

6.3 Translating Local Access to Variables

Local variables are variables declared in a procedure and accessed within the body of the procedure itself. Aside from variables that are declared in the declaration part of the block, there are the parameters and function identifier which in PASCAL are treated like local variables.

Translations of local variables depend on the relative position of their storage in the stack. We can illustrate how translation can proceed by considering the following PASCAL procedure and its translation:

```
function local(a: integer;var b: integer): integer;   @1: D2 M1 M1
   var c: integer;                                           M1
       d: integer;                                           M1
```

Our picture of the stack after a call to this procedure and execution of its declaration part is shown in figure 6.4

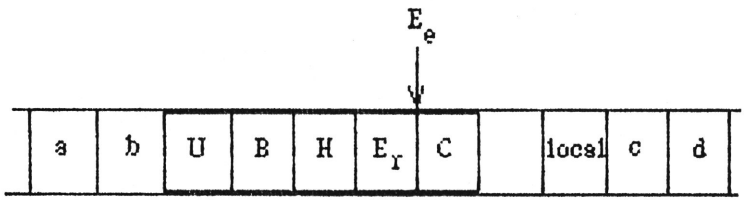

Fig. 6.4 The stack after the execution of the declaration of function 'local'.

The local environment of function 'local' is E_e, so that all accesses to local variables will use the value of E_e.

The SLIM link in figure 6.4 consists of 5 cells. The B entry contains a pointer to the name of the procedure. It is particularly useful when an error occurs because it provides the names of the procedures called before the error. In the succeeding chapters we shall be assuming a SLIM implementation with 5 cells in the link (similar to the SLIM link in figure 6.4) and a stack that grows backward (unlike the stack in figure 6.4 which grows forward).

To illustrate how to access the variables then, we continue the example

```
begin
  local :=
          a          LIE-5       P
```

```
            + b          LIE-4 LI0   +H P
            + c          LIE3        +H P
            + d          LIE4        +H
                         SE2
end;                     LIE2
```

What we should note in this example is the translation of the parameter 'b'. The translation is quite different because 'b' was declared as a variable (var) parameter and therefore what is passed to the cell allocated to 'b' is the address of the cell where it is stored. Hence, loading the value of 'b' in the accumulator means loading the cell address where it is stored and indexing by 0, i.e., LIE-4 LI0.

Translation of local variables of array and record type are carried out in the same way. To illustrate the translation, we use the same example, but with all the variables and parameters declared as an array with one element, i.e., array[1..1] of type.

Again, we show in figure 6.5 the picture of our stack after the execution of the declaration part is complete. Each array in the stack occupies 5 cells (1 cell for the element of the array and 4 cells for the dope vector).

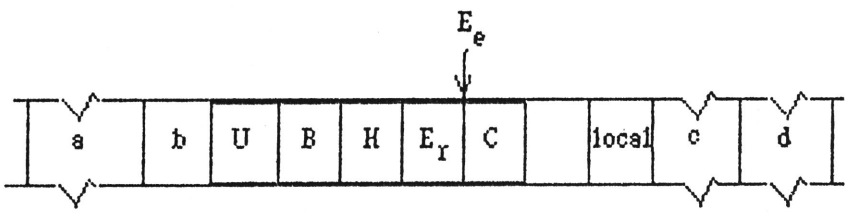

Fig. 6.5 The stack after the execution of the declaration of function 'local'. The variables a, b, c, and d are declared as an array.

Consider an example statement and its translation below.

```
begin
  local :=
          a[1]    LE-10  SE1 P LI0 +H P L0 P L1 +H +H LI0  P
        + b[1]    LIE-4  SE1 P LI0 +H P L0 P L1 +H +H LI0  +H P
        + c[1]    LE3  SE1 P LI0 +H P L0 P L1 +H +H LI0   +H P
        + d[1]    LE8 SE1 P LI0 +H P L0 P L1 +H +H LI0    +H
                  SE2
end;              LIE2
```

It should be observed that, in the translation of the variables, the first instructions reflect the kind of local variable, i.e., parameter, function identifer, and locally declared variable.

6.4 Translating Non-local Access to Variables

Variables declared in a procedure and accessed in procedures nested in it are called non-local variables. Usually, access to this variable requires at least more than one instruction. In extended SLIM, accessing a non-local variable is basically the same as accessing a local variable, except that you must specify which E register to use. Specifying which E register is done by generating a U instruction taking as operand the difference in textual level of the procedure where the variable is declared and the procedure where it is accessed. The rest of the translation is similar to local variables.

We, again, appeal to an example to illustrate how the translation of a non-local variable can be carried out. Consider the following procedure declarations and their translation

PASCAL	SLIM	
procedure one;	$"one"	@1: DO M1
var a1,b1: integer		M1 M1
procedure two;	$"two"	@2: DO M1
var a2,b2: integer;		M1 M1
procedure three;	$"three"	@3: DO M1
var a3,b3: integer;		M1 M1
begin		
a3 :=		
a2		U1 LIE2 P
+ b1;		U2 LIE3 +H
		SE2
end;		
begin		
end;		
begin		
end;		

Of importance is the translation of the variable 'b1'. Note that it is declared in procedure 'one' and is accessed in procedure 'three'. Since the textual level of procedure 'one' is two levels shallower than procedure 'three', therefore the operand of the U instruction that is necessary to access variable 'b1' is 2. Hence the translation U2 LIE3. Note that a negative operand to the instruction U is impossible.

CHAPTER 7

TRANSLATING PASCAL EXPRESSIONS

PASCAL expressions are constructs denoting rules of computation for obtaining values by application of operators. Its primary constituents are constants, variables, elements of a data structure, or results of function calls.

Typical PASCAL expressions are arithmetic expressions and boolean expressions. Arithmetic expressions are expressions whose results are numbers. Number results can be integers or real numbers. Most often, arithmetic expressions are used to set the value of a variable, i.e., used in assignment statements. A boolean expression, on the other hand, yields a boolean value, i.e., either true or false. Boolean expressions are used by most of PASCAL's basic control constructs like repeat statements, while statements, etc. They are sometimes used in assignment statements to set the value of a boolean type identifier.

7.1 Operator Precedence

As in other high-level languages, PASCAL expressions obey operator precedence. In cases where an expression is composed of arithmetic and boolean expressions, arithmetic operators always have higher precedence over boolean operators. Moreover, arithmetic and boolean operators also follow precedence rules among themselves. We summarize PASCAL's precedence rules by listing the operators in order of descending precedence.

Operators		Priority	Description
( )		1	Grouping operator
Arithmetic Operators			
Integer	Real		
-	-	2	Negate operator
*, div, mod	*, /	3	Multiplying operators
+, -	+, -	4	Adding operators
Boolean Operators			
not		5	Logical not operator
and		6	Logical and operator
or		7	Logical or operator
=, <>, <, >, <=, >=		8	Relational operators

Operators having the same priority are evaluated from left to right.

7.2 Description of the General Method Used by the Translator

The translation of expressions to efficient object code has always been the objective of most compiler writers. But the fullfillment of this objective depends mostly on two factors. These factors are the design of the translator and the number of registers available in the target machine that can be used for evaluating expressions.

Consider two translators of different designs: the first is a translator that builds a parse tree and generates object code from the said tree and the second is a translator that generates object code directly from the source code. The first type of translator has some knowledge of the form of the expression that it has to translate. This knowledge enables the translator to select an optimal path in the tree so as to generate the most efficient object code for the expression. In this type of translator,

however, extra time is spent on building the parse tree and touring it. The second type of transaltor avoids the overhead of building and touring the parse tree. But, this means having no knowledge of the form of the expression. Usually, the translator has to contend with the generation of object code that works for all cases. Further, code generated this way is usually sub-optimal, although it can also generate optimal code but only for very simple expressions.

The availability of registers in the target machine that can be used for evaluating expressions is another factor for generating efficient object code for expressions. Usually, the rule is - the more registers available in the target machine, the more efficient the translation of expressions can be. With more registers available, translation could proceed by dividing the expressions into as many sub-expressions as possible and evaluating each sub-expression using one separate register. This avoids the overhead of storing the result of one sub-expression to the memory to allow evaluation of the next sub-expression.

Unfortunately, our translator and target machine is very different from the ideal translator and target machine described for generating efficient object code for expressions. The translator, as described earlier, is a one-pass translator generating code directly from the source code. The target machine, SLIM, has only one register, the accumulator, available for evaluating expressions.

The Reverse Polish method of evaluating expressions seems to be the most appropriate method to use for the translator described and SLIM as the target machine. The stack and the accumulator is used to hold values of the expression's constituents to enable expressions to be converted to Reverse

Polish form.

Some examples of expressions and their Reverse Polish form are given below.

Expression	Reverse Polish Form
a - b + c * d / e	a, b, -, c, d, *, e, /, +
(a + b) * (c / d)	a, b, +, c, d, /, *
a * b > c - d	a, b, *, c, d, -, >

The method works similar to the shunting method in a railway network. It moves the contitutuents of the expression to the stack and remembers the operator in order. Every time the translator receives an identifier from the scanner, it decides whether to evaluate the sub-expression using the value in the accumulator (where the value of the identifier is) and the value on top of stack as operand, or to store the value of the identifier for later evaluation. If it decides to evaluate, it uses the latest remembered operator to evaluate the values in the accumulator and the top of stack delivering the result in the accumulator. The top of stack, of course, is decremented by one. In case it decides to store the value of the identifier (because the operator following it is of lower precedence), then it pushes the value in the accumulator into the stack and loads the value of the identifier into the accumulator.

Perhaps, the method will become clearer after investigating where the values of the constituents of an example expressions, a+b*c=c+d, during translation are stored.

Expression	Accumulator	Stack	Operator
a	a		
+	a		+

b	b	a	+
*	b	a	*, +
c	b*c	a	+
	a+b*c		
=	a+b*c		=
c	c	a+b*c	=
+	c	a+b*c	+, =
d	c+d	a+b*c	=
	a+b*c=c+d		

7.3 Translating Arithmetic Expressions

We have discussed the general method of translating expressions. Now, we shall see how this method fits with the actual translations of PASCAL expressions to SLIM.

Suppose 'a' and 'b' are the first two, in that order, local variables of a procedure. The translation of some simple arithmetic expressions involving 'a' and 'b' and one of the arithmetic operators can be summarized as follows:

Commutative Operators	SLIM
a + b	LIE2 P LIE3 +H
a * b	LIE2 P LIE3 *H

Non-commutative Operators	SLIM
a - b	LIE2 P LIE3 PLH -H
a div b	LIE2 P LIE3 PLH /H

To illustrate the translation of more complex expressions involving 'a' and 'b', we show the example expresson, a + b div a - b * a, and its translation.

Expression	Translation
a	LIE2
+	P
b div a	LIE3 P LIE2 PLH /H
	+H
-	P
b * a	LIE3 P LIE2 *H
	PLH -H

The constituents of an arithmetic expression may assume real and integer values. In this case, PASCAL follows some simple rules of resolving the type of the final result of the expression. We summarize these rules by listing the possbible values of the operands and the result of each expression involving these operands.

Left Operand	Right Operand	Result
Integer	Integer	Integer
Real	Integer	Real
Integer	Real	Real
Real	Real	Real

The SLIM instruction *: (FLOAT) is used to conform with PASCAL's rules on result types. Below are examples for each possible values of the operands of an arithmetic expression and how the SLIM instruction *: (FLOAT) is used.

Expression	SLIM
10 + 10	L10 P L10 +H
10.0 * 10	L10.0 P L10 *: *+H
10 / 10.0	L10 P L10.0 PLH *: */H
10.0 - 10.0	L10.0 P L10.0 PLH *-H

There are cases when the operands are integer type and the expression is expected to be of real type. For example

a := 10 + 10

where 'a' is declared to be real, this can be resolved by adding SLIM's *: (FLOAT) operator at the end of the expression, i.e.,

L10 P L10 +H *: SE2.

7.4 Translating Boolean Expressions

Boolean expressions are expressions involving relational operators or logical operators. Expressions involving relational operators are translated in a similar manner to arithmetic expressions. But, expressions involving logical operators are usually translated differently to take advantage of the fact that the value of a boolean expression can be established by knowing the value of at least one of its sub-expressions.

Suppose 'a' and 'b' are the first and second local variables of a procedure. First, we shall summarize the translation of boolean expressions involving 'a' and 'b' and one of the relational operators.

Relational Operators	SLIM
Commutative	
a = b	LIE2 P LIE3 =H
a <> b	LIE2 P LIE3 ~=H

Non-commutative	
a < b	LIE2 P LIE3 PLH <H
a > b	LIE2 P LIE3 PLH >H
a <= b	LIE2 P LIE3 PLH <=H
a >= b	LIE2 P LIE3 PLH >=H

What we should observe from this summary is that the translation given for non-commutative relational operators will not always be correct when the type of 'a' and 'b' are boolean. Note that SLIM uses 0 to represent false and -1 to represent true and therefore a true value is less than a false value. PASCAL, on the other hand, uses 1 to represent true which makes a true value greater than a false value. Using the translations above, we shall run into trouble when 'a' or 'b' assume opposite values. For example, if 'a' is true and 'b' is false, the result of the expression 'a < b' is true, where in fact it should be false since true is greater than false. To avoid the problem, special code is generated when the type of the operands is boolean. Instead of the previous translation, the following are generated for boolean expressions involving non-commutative relational operators with boolean operands.

Non-commutative Relational Operators	SLIM
a < b	LIE2 - P LIE3 - PLH <H
a > b	LIE2 - P LIE3 - PLH >H
a <= b	LIE2 - P LIE3 - PLH <=H
a >= b	LIE2 - P LIE3 - PLH >=H

The new special translation negates the values of the operands. This would make all true values to be in accordance with the PASCAL specification, i.e., true is 1. It would not, however, affect the false values, since the negative of 0 is still 0.

Next, we consider the translation of boolean expressions involving logical operators. As mentioned earlier, translaton of this type of boolean expressions is handled differently to reduce execution time when the value of at least one sub-expression is known. The common way of doing this is to include jumping code to skip part of the expression that does not change the final value of the expression. In the case of PASCAL, code to skip part of an expression can only be done when the boolean expression involves the logical 'and' or logical 'or' operators. The PASCAL specification states that boolean expressions involving a logical 'or' operator yield a true value if either or both of the operands are true. Boolean expressions involving a logical 'and' operator, on the other hand, yield true if both the operands are true. Below we show a summary of expressions involving logical operators and their translation.

Logical Operators	SLIM
a and b	LIE2 F@1 P LIE3 /\H @1:
a or b	LIE2 T@1 P LIE3 \/H @1:
not a	LIE2 ~

Note that the translation includes the jumping code. But when 'b' is a function call, the jumping code, i.e.,

F@label ... @label: or T@label ... @label:,

is not generated. This must be done because a function call may produce a side effect, e.g., it may modify the value of a non-local variable, such that execution of it is necessary. Hence, in this case, i.e., the second operand of the logical expression is a function call, translation will be similar to boolean expressions involving relational operators.

CHAPTER 8

TRANSLATING PASCAL STATEMENTS

PASCAL statements can be divided into two groups according to their composition: simple and structured statements. Simple statements are constructs containing no other statements; while structured statements are statements which may be executed in sequence, e.g., compound statements, conditionally, e.g., IF and CASE statements, and repeatedly, e.g., WHILE, REPEAT, and FOR statements.

This chapter will show the translation of PASCAL statements to SLIM. It will discuss in detail the translation of assignment statements, procedure statements, IF statements, CASE statements, WHILE statements, REPEAT statements, and FOR statements.

8.1 Translating Assignment Statements

Assignment statements in PASCAL take the general form

destination := source.

The translation of assignment statements involves the calculation of the address of the destination and value of the source. Computing the value of the source may also involve an address calculation. The computation of addresses when the destination of an assignment is of array and/or record type were discussed in Chapter 6. What this section aims to do is to present a more specific example to illustrate most of the forms of assignment statements and how they can be improved at compile time.

Consider the declaration

```
type recd = record
              r1: integer;
              r2: integer;
            end;
var  a: integer;
     b: integer;
     c: array[1..1] of integer;
     d: recd;
```

Assuming local access, some of the legal assignment statements that can be constructed from the declaration and their corresponding translation is summarized below.

PASCAL	SLIM
a := b	LE2 P LIE3 SH
a := c[1]	LE2 P LE4 SE1 P LI0 +H P L0 P L1 +H +H LI0 SH
a := d.r2	LE2 P LE9 -1 LI0 SH
c[1] := d.r2	LE4 SE1 P LI0 +H P L0 P L1 +H +H P LE9 -1 LI0 SH
d.r2 := c[1]	LE9 -1 P LE4 SE1 P LI0 +H P L0 P L1 +H +H LI0 SH

One thing noticeable about the translation of assignment statements above is the important role the top of stack plays in the translation. It is used to hold results of the computation of the address of the destination and the value of the source (which also involve the computation of the address) of the assignment.

The translation of assignment statements having an entire variable as destination, e.g., a := b, can be improved at compile time. Note that the translation of the destination of the assignment

```
LE2 P ... ... ... SH
```

can also be written as

```
... ... ... SE2.
```

Incorporating this improvement in the example, we then get

PASCAL	SLIM
a := b	LIE3 SE2
a := c[1]	LE4 SE1 P LI0 +H P L0 P L1 +H +H LI0 SE2
a := d.r2	LE9 -1 LI0 SE2

which saves two instructions per assignment statement.

Assignment statements involving subscripted variables like

```
c[1] := d.r2
```

will require a more complicated optimization technique to get a reasonable improvement. One technique is discussed in chapter 10.

8.2 Translating Procedure Statements

Procedure statements are commonly known as procedure calls. They are statements that serve to execute the procedure. A procedure statement consists of a procedure name and, in some cases, actual parameters. Actual parameters, if they are part of the call, should have a matching type and number with that of the formal parameters of the procedure declaration.

PASCAL supports three kinds of procedure calls; call to a user-defined procedure or function, call to a standard procedure, and call to a standard function. In the succeeding sections, translation of each kind of call will be discussed in detail.

8.2.1 Translating Calls to User-Defined Procedures

A call to a user-defined procedure may be made with or without parameters. The presence of parameters, of course, depends on how the procedure was declared, i.e., if it was declared with one formal parameter then it should be called with one actual parameter.

First, we shall consider how a call with no parameters is translated. This translation will have to proceed with only the procedure name and information in the symbol table to work with.

Since PASCAL supports non-local variables, it has to set the value of its static environment before the actual call is made. This value, the static environment, can be set by knowing the difference between the textual level of the procedure where the call is made (caller) and the textual level of the called procedure (callee). Assuming there are five (5) cells in the stack link, the static environment of the procedure can be set by the user by generating one of the following SLIM instructions.

Difference in textual level (caller - callee)	SLIM
-1	U0 LE0
0	LIE-4
k (k>0)	Uk LIE-4

We mentioned in Chapter 3 that UO is an instruction that does nothing. In the case where the difference in textual levels is -1, LEO should have been enough to set the the static environment of the procedure. But, LEO alone will load a cell address and will cause problems in byte addressable machines where the static environment is stored as a machine address. Extended SLIM uses the UO instruction to signal that the value to be loaded is the machine address of the leftmost byte of the cell. It should therefore be generated by the compiler.

After the static environment of the procedure has been set, the SLIM call (C) instruction follows. The call instruction can take two different operands depending on how the procedure is declared. The call instruction may take the form

C@entry or CIE-k.

In the first form, @entry is the entry point of the procedure. It is the same as the label generated during procedure declaration, i.e.,

```
procedure test;                $"test" @entry: DO M1.
```

The second form of calling a procedure is used only when calling a procedure that was passed as a parameter. The E-k is the address of the cell where the entry point of the procedure is stored. The offset is negative because it was passed as a parameter and thus is found before the SLIM link.

Finally, SLIM requires that the number of actual parameters be

supplied during a procedure call. The SLIM (pseudo)instruction

DX8000

can be used to supply this value, where X8 (X means that the number is in hexadecimal format) is used by SLIM to indicate that the static environment is set by the user and 000 (also in hexadecimal) indicates the number of actual parameters.

Accordingly we consider the following procedure declarations and calls with their corresponding translations to illustrate how calls with no parameters are translated.

PASCAL	SLIM
procedure one;	$"one" @1: DO M1
procedure two;	$"two" @2: DO M1
procedure three;	$"three" @3: DO M1
begin	
one;	U2 LIE-4 C@1 DX8000
two;	U1 LIE-4 C@2 DX8000
end;	
procedure threeb;	$"threeb" @4: DO M1
begin	
three;	LIE-4 C@3 DX8000
end;	
begin	
three;	U0 LE0 C@3 DX8000
end;	
begin	
end;	

The call to a procedure with parameters is similar to a call with no parameters except that the parameters have to be pushed first onto the

stack. The values of the actual parameters to be passed will depend on the type of the corresponding formal parameters. The parameter may be a value parameter, variable (var) parameter, or a procedure parameter.

As the name implies, value parameters pass the value of the variables. This means that if the parameter is declared as an array or a record the whole data structure is copied to the parameter area (in the stack) of the procedure.

The second type of parameter is the variable (var) parameter. Variable (var) parameters pass only one value to the called procedure. This value is the address of the cell where the variable is stored. If, however, the parameter was declared as an array or record then only the address of the first cell allocated to the data structure is passed.

Finally, a parameter may be a procedure parameter. In which case the static environment and the procedure entry is passed to the called procedure.

The data instruction following the call instruction must indicate the number actual parameters. For example, a call to a procedure with 13 parameters will require the data instruction DX800D.

Now, consider an example where all the types of parameters occurs and their translation.

PASCAL SLIM

```
program parameters(output);
type vector = array[0..1] of integer;
```

```
var i: integer
    a: vector;
procedure pass(                                  $"pass" @3: D11 M1
                  k: integer;
                  x: vector;
              var l: integer;
              var y: vector;
              procedure  parm);
  begin
    parm;                                         LIE-5 CIE-6 DX8000
  end;
 procedure passme;                               $"passme" @4: D0 M1
  begin
  end;
 begin
  pass( i,                                        LIE2 P
        a,                                        LE3 SE1
                                                   LIE1 LI0 P LIE1 LI1 P
                                                   LIE1 LI2 P LIE1 LI3 P
                                                   LIE1 LI4 P LIE1 LI5 P
        i,                                        LE2 P
        a,                                        LE3 P
        passme);                                  LI@9 U0 PLE0 P

                                                U0 LE0 C@3 DX800B
 end.                                             $: @9: D@4
```

Note from this example how calls to a procedure passed as a parameter, i.e., parm, are translated.

8.2.2 Translating Calls to Standard Procedures

Most of PASCAL's standard procedures deal with input and output. The actions of some of these standard procedures can already be handled by some of the existing SLIM library routines. For other standard procedures which are specific to PASCAL, the best way of making them available is to write

these procedures and incorporate them in the SLIM library routines. This approach will make the translation of standard procedures uniform. The same approach will be considered in the discussion below.

The call to a standard procedure has a slight difference from that of a call to a user-defined procedure. In a call to a standard procedure, the user does not bother about setting the static environment of the procedure. This means a translation of at least one instruction less than the translation of user-defined procedures. The entry points of these procedures are globally known and their values (of entry points) can be accessed through the SLIM G register.

The general format of the call to a standard procedure is

$$Lp_1\ P\ Lp_2\ P\ \ldots\ Lp_n\ CIGk\ Dn$$

where p_i's are the parameters, Gk is the address of the cell where the entry point of the procedure is stored and n is the number of actual parameters. The (pseudo)instruction that supplies the information about the number of actual parameters, i.e., Dn, does not contain the flag 'X8' and thus SLIM knows that the value in the accumulator is a parameter and not the value of the static environment. It is interesting to observe that these reduce to the standard procedure call used when translating BCPL to SLIM.

Again, we shall appeal to an example to have a broad sweeping view of the translation involving calls to standard procedures. Consider the following program and translations of the standard procedures in it.

PASCAL	SLIM
`program stndproc(input, output);`	
`const wd = 12;`	
`dp = 3;`	
`var i: integer;`	
`r: real;`	
`c: char;`	
`b: boolean;`	
`begin`	
`readln( i,`	L1 CIG84 D1 SE2
`c,`	L3 CIG84 D1 SE4
`r);`	L2 CIG84 D1 SE3
	L5 CIG84 D1
`writeln(' string ',`	LI@4 CIG65 D1
`b,`	LIE5 PL10 CIG66 D2
`i,`	LIE2 PL10 CIG71 D2
`i: wd`	LIE2 PL12 CIG71 D2
`c,`	LIE4 CIG60 D1
`r,`	LIE3 PL15 PL6 CIG69 D3
`r: wd: dp);`	LIE3 PL12 PL3 CIG69 D3
	CIG67 D0
`end.`	$ @4: D" string "

The example shows that the translation of standard procedures with several parameters is being done by translating its equivalent set of statements with one parameter. Note that in PASCAL, the statement

```
readln(i, c, r);
```

has the same actions as the set of statements

```
read(i);
read(c);
```

```
read(r);
readln;.
```

The translation, if one noticed, is much like this equivalent set of statements.

Another important observation from the example is the use of default values in the translation of the output statement writeln. Note for example, the translation of the part that writes a real, i.e., write(r), the translation introduces the default field width of 15 and number of decimal places of 6. These values, according to PASCAL specification, are implementation dependent and are the implementor's choice.

One of the dangers in using numbers to represent standard procedures is that they may not be the same from one SLIM implementation to another. So, since most of these routines are written in BCPL, we show the equivalents of these PASCAL's standard procedures in BCPL hoping that the equivalent name in BCPL does not change.

```
     PASCAL            BCPL                    Comment

readln( i,          i := pread(1);       // read an integer
        c,          c := pread(3);       // read a character
        r);         r := pread(2);       // read a real
                    pread(5);            // skip until newline character

writeln(' string '  writes(" string "); // write a string
       b,           pwritb(b, 10)        // write a boolean
       i,           writed(i, 10);       // write an integer
       i: wd        writed(i, 12);
       c,           wrch(c);             // write a character
       r,           writefp(r, 15, 6);   // write a real
       r: wd: dp);  writefp(r, 12, 3);
                    newline();           // write a newline character
```

The procedures that start with 'p' are PASCAL specific procedures. They have to be written and incorporated into the SLIM library routines. The procedures which start with 'p' also use some of the existing SLIM library routines.

The example given above does not show the output incompatibility between PASCAL and SLIM. SLIM will only flush the output buffer when it encounters a write a newline character and if one does not come, it outputs nothing to the screen. PASCAL, on the other hand, has to flush the output buffer everytime a write statement is encountered. To illustrate the problem, consider the same example, but this time using a 'write' statement instead of 'writeln'. To avoid the problem caused by the output incompatibility between PASCAL and SLIM, the translation of a 'write' statement should be followed by an instruction which will flush the output buffer. Hence, the translation of the same example with 'write' statement instead of 'writeln' is

```
write(' string ',          LI@4 CIG65 D1
      b,                   LIE5 PL10 CIG86 D2
      i,                   LIE2 PL10 CIG71 D2
      i: wd                LIE2 PL12 CIG71 D2
      c,                   LIE4 CIG60 D1
      r,                   LIE3 PL15 PL6 CIG69 D3
      r: wd: dp);          LIE3 PL12 PL3 CIG69 D3

                           CIG5 D0

                           $ @4: D" string "
```

in BCPL, this will be

```
write(' string '    writes(" string ");   // write a string
    b,              pwritb(b, 10)         // write a boolean
```

```
i,              writed(i, 10);      // write an integer
i: wd,          writed(i, 12);
c,              wrch(c);            // write a character
r,              writefp(r, 15, 6);  // write a real
r: wd: dp);     writefp(r, 12, 3);
                flush();            // flush the buffer
```

8.2.3 Translating Calls to Standard Functions

Due to the variety of standard functions available in PASCAL, it is impossible to have a common format of translating them. Standard functions supported by PASCAL may return a real, integer, character, or boolean value.

Among the standard functions, that need special attention is the translation of mathematical functions. Mathematical functions in PASCAL includes the sin, cos, arctan, ln, exp, and sqrt functions. The translation of these functions can be approached in several ways. One way is to write the functions themselves and incorporate them in the SLIM library routines. The translation will then be the same as standard procedures. But, if your machine has existing mathematical libraries that compute these functions, then a routine can be written to make these machine's library routines accessible. This routine can then be incorporated in the SLIM library routines and therefore translation can proceed as for a standard procedure. An example of translation using the math functions in the C library routines in a UNIX machine is given below.

Example:

PASCAL	SLIM	BCPL
sin(x)	Lx P L1@4 PL1 C1G28 D3	fcallc(x, "sin", 1)

```
... ... ...
$ @4: D"sin"
```

The rest of the standard functions, except the end of file (eof) and end of line (eoln) functions, can be translated using SLIM instructions that do not involve a call to the SLIM library routines. These translations are summarized below.

Standard Function	SLIM
abs(x)	Lx *|
sqr(x)	Lx P **H
abs(n)	Ln |
sqr(n)	Ln P *H
trunc(x)	Lx *.
round(x)	Lx *+0.5 *.
odd(n)	Ln /*2 -
eoln(input)	C1088 D0
eof(input)	C1089 D0
ord(c)	Lc
chr(n)	Ln
pred(c)	Lc -1
succ(c)	Lc +1

The variables x, n, and c are of real, integer, and character type respectively. The SLIM instruction *. (FIX) is the same as the FIX instruction in BCPL and TRUNC instruction in PASCAL. Its action is to truncate the decimal part of a real value and convert the resulting value to its integer equivalent. Note the translation of ord(c) and chr(n), nothing extra is generated because characters are internally represented by their integer equivalents. Bounds checking for chr(n), pred(c), and succ(c) were omitted. The check can be done by generating C1087 D0 after the

translation given above for the functions. The set of instructions C1087 DD is a call to a routine in the SLIM library routines that handles error checking.

8.3 Translating IF Statements

The decision primitive in PASCAL is the IF statement. It is a conditional statement that executes a constituent statement only if a certain condition, usually a boolean expression, is true. If the condition is false, it either executes no constituent statement or executes the constituent statement following the 'else' keyword.

There are two basic formats for translating IF statements. One occurs when the 'if' keyword does not have a matching 'else', i.e.,

```
if (expression) then (statement);
```

in which case the translation can proceed as

```
(expression) F@k (statement) @k:
```

The next, of course, is when the 'if' keyword has a matching 'else', i.e.,

```
if (expression) then (statement 1)
else (statement 2);
```

where the translation is

```
(expression) F@k (statement 1) J@l
```

```
        @k: (statement 2)
        @1:
```

Now, consider the following example for each case. In the example, assume 'i' to be the first local variable of a procedure.

PASCAL	SLIM

`if i > 0 then`
`  i := 1;`
	`LIE2 P LO PLH >H F@1`
	`  L1 SE2`
	`@1:`

`if i > 0 then`
`  i := 1`
`else`
`  i := -1;`
	`LIE2 P LO PLH >H F@1`
	`  L1 SE2`
	`J@2: @1:`
	`  L-1 SE2`
	`@2:`

8.4 Translating CASE Statements

The second conditional statement is the CASE statement. It can be viewed as a multiple IF statement. In contrast to the IF statement that has a choice of at most two alternatives, CASE statements can have more. Specifically, a CASE statement is a conditional statement that specifies that the constituent statement whose label is equal to the current value of the expression in the beginning of the statement be executed.

The general format of a CASE statement and its translation is

PASCAL	SLIM
`case (expression) of`	`(expression) P J@1`
`c11, ..., c1m: (statement 1);`	`@11: @12: ... @1m: (statement 1) J@4`
`c21, ..., c2m: (statement 2);`	`@21: @22: ... @2m: (statement 2) J@4`

```
  c31, ..., c3m: (statement 3);   @31: @32: ... @3m: (statement 3) J@4
     ...                                  ...
     ...                                  ...
     ...                                  ...
  cn1, ..., cnm: (statement n);   @n1: @n2: ... @nm: (statement n) J@4
end;                              @3: L1 CIG87 D1 Q   // send an error
                                                      //  and quit
                                  @1: L(m*n+1) ?S
                                         D@3          // default label
                                   Dc11  D@11
                                   Dc12  D@12
                                      ...
                                   Dc1m  D@1m
                                   Dc21  D@21
                                   Dc22  D@22
                                      ...
                                   Dc2m  D@2m
                                      ...
                                      ...
                                   Dcn1  D@n1
                                   Dcn2  D@n2
                                      ...
                                   Dcnm  D@nm
                                  @4:
```

Now, consider the following example to see how this general format of a CASE statement applies to a specific PASCAL program. Assume that 'i' is the first local variable of a procedure.

```
        PASCAL                          SLIM

    case i of                       LIE2 P J@1
      1, 2:  i := 2;                 @3: @4: L2 SE2 J@2
      3, 4:  i := 4;                 @5: @6: L4 SE2 J@2
        10:  i := i div 10;              @7: LIE2 P L10 PLH /H SE2 J@2
    end;                                 @8: L1 CIG87 D1 Q

                                    @1: L6 ?S
```

```
        D@8
   D1   D@3
   D2   D@4
   D3   D@5
   D4   D@6
   D10  D@7
@2:
```

8.5 Translating WHILE Statements

One of the structured statements that executes a set of statements repeatedly is the WHILE statement. Its general format is

while (expression) do (statement).

It continuously executes the statement as long as the expression, which is usually a boolean expression, is true. Note that the expression is evaluated first. Obviously, the values of the constituents of the expression must be modified within the statement if execution is to terminate.

The general format of the SLIM translation of a WHILE statement is

@k: (expression) F@1 (statement) J@k @1:.

Again, consider the following example and its translation. Assume that 'i' is the first local variable of a procedure.

PASCAL	SLIM
while i < 10 do	@1: LIE2 P L10 PLH <H F@2:

```
i := i + 1;                    LIE2 P L1 +H SE2 J@1
                               @2:
```

We should also observe from the statement and its translation that it is possible for the statement to be not executed at all. In short, the number of iterations can be zero.

8.6 Translating REPEAT Statements

In constrast to WHILE statements, which evaluate the boolean expression before executing the statement, a REPEAT statement executes the statement first before evaluating the boolean expression. This means that REPEAT statements execute the statement at least once. Another basic difference between REPEAT and WHILE statements is the value of the boolean expression for the iteration to continue. In REPEAT statements, if the value of the boolean expression is false then iteration continues. But in WHILE statements, iteration continues only when the value of the boolean expression is true.

The general format of a REPEAT statement and its translation is

PASCAL	SLIM
repeat (statement);	@k: (statement)
until (expression)	(expression) F@k.

Again consider the following example, where 'i' is the first local variable, and its translation.

PASCAL	SLIM
repeat	@1:

```
    i := i + 1;                    LIE2 P L1 +H SE2
until i > 10                       LIE2 P L10 PLH >H F@1.
```

Note that for the statement to terminate it should also modify the contituents of the boolean expression inside the statement. Of course, except when the condition is already true before the start of the execution of the REPEAT statement.

8.7 Translating FOR Statements

The FOR statement is similar in effect to both REPEAT and WHILE statements, except that it produces a predetermined number of iterations each time it is executed.

The general format of a FOR statement is

```
for control := initial to final do (statement);
```

or

```
for control := initial downto final do (statement);
```

where 'control' is a declared variable, 'initial' and 'final' are arithmetic expressions. The 'to' format increments the control variable by 1 while that of the 'downto' format increments it by -1. Further, the value of 'initial' and 'final' should be evaluated only once and should not be altered by the repeated execution of the statement.

Now, we show the SLIM translation of FOR statements

PASCAL	SLIM
for (control) := (initial) to (final) do (statement);	(initial) S(control) (final) J@1 @2: (statement) L(control) +1 S(control) LH @1: P >=(control) T@2 M-1

The translation of the 'downto' format is the same, except that 'control' is decremented instead of incremented, i.e., L(control) -1 S(control) instead of L(control) +1 S(control) and the comparison is reversed, i.e., <=(control) instead of >=(control).

Again, consider the following example, where 'i' is the first local variable, and its translation.

PASCAL	SLIM
for i := 1 to 10 do write(i);	L1 SE2 L10 J@1: @2: LIE2 PL10 CIG71 CIG5 DO LIE2 +1 SE2 LH @1: P >=IE2 T@2 M-1

One important observation from the translation is that the value of 'control' after the execution of the FOR statement is equal to 'final'+1. It should not bother the implementor since in standard PASCAL the value of 'control' is undefined on exit from the FOR statement or even when a goto statement is used to force the termination of the FOR statement.

CHAPTER 9

CREATION AND USAGE OF THE SYMBOL TABLE

The symbol table is important in the operation of a compiler. Its main purpose it to serve as a library of information needed by the compiler during several stages of compilation. It is used during lexical analysis to search for identifier names and check whether these identifiers are consistent with the declaration. During translation, it is the source of information which determines the kind of code to be generated. It also is responsible for providing information about the amount of storage needed to be allocated for a particular variable.

This section, however, will be concerned only with the information stored in the symbol table and its use during translation. The discussion will be based on the following example declaration.

```
const konstant = 100;
type enumtype = (mon, tue, wed);
       rekord = record
            r1: integer;
            r2: array[-1..2] of char;
             r3: real;
       end;
var  int: integer;
     root: real;
     bool: boolean;
     ch: char;
     table: array[0..2] of array[1..3] of integer;
     recd: rekord;
     arrecd: array[3..5] of rekord;
procedure pass(valuep: integer
               var varp: integer;
               procedure procp(procedure inprocp));
```

9.1 Contents of the Symbol Table

The symbol table consists of two parts: the identifier name and the information about the identifier. The information about the identifier necessary during translation of PASCAL to SLIM contains the following fields: identifier type (id_type), element type (element_type), array or block reference (arblk_ref), kind of parameter (p_type), textual level (level), and offset from the environment register (offset). Of course, not all the fields of the information about an identifier will be used during translation. Some identifier types may use all the information fields in the table about itself but there are some identifiers that uses only one or two information fields to get translated. We shall be presenting an entry in the table about the identifier in the following format

identifier	id_type	element_type	arblk_ref	p_type	level	offset
...	...	...	...	...	...	...

In addition to the table just described, are two tables that keep information about arrays and blocks. The additional table on arrays includes the following fields of information: array number (arrays), element type (element_type), array or block reference (arblk_ref), lower bound (low), upper bound (high), element size (elm_size), total size (total_size), and the size of the dope vector (dope_v_size). The table on block information, on the other hand, includes the fields: block number (blocks), parameter size, and variable size. These two tables can be accessed using the array or block reference (arblk_ref) field of the main table.

9.2 Creation and Usage of Information from Constant Definitions

Since all occurences of the constant identifier will be replaced by the value it denotes, the symbol table then should contain the value of the identifier. Referring to the example declaration, consider the constant declaration

```
konstant = 100;.
```

The corresponding entry in the symbol table created for this definition is

identifier	id_type	element_type	arblk_ref	p_type	level	offset
konstant	constant	integer	0	0	1	100

What can be observed from the information in the table is that the value of the identifier is stored in the 'offset' field. This means that the translation of constant identifiers can be carried out by accessing the value of the 'offset' field of the entry corresponding to the identifier. For example, consider the code fragment

```
1 + 2 * konstant
```

and its translation

```
L1 P L2 P L100 *H +H.
```

Note in the example that the translation of identifier 'konstant' is carried out by replacing it with the value stored in the 'offset' field.

9.3 Creation and Usage of Information from Type Definitions

The example declaration we considered involves two kinds of type definitions. The first is the definition of an identifier as an enumerated type, i.e.,

```
enumtype = (mon, tue, wed);
```

The entries in the symbol table created for this kind of definition are

identifier	id_type	element_type	arbik_ref	p_type	level	offset
enumtype	type	enumtype	0	0	1	1
mon	constant	enumtype	0	0	1	0
tue	constant	enumtype	0	0	1	1
wed	constant	enumtype	0	0	1	2

Note the identifier type of the elements of the enumeration. Observe that enumerated type elements are considered as constants and their values, which are in the 'offset' field, are based on their position in the enumeration. The first position having a value zero. This suggests that code fragments like

```
mon > tue
```

can be translated using the information in the symbol table as

```
L0 P L1 PLH >H.
```

The second kind of type definition in the example is the record type definition. A record type definition stores information about the record in

the symbol table. It stores the amount of storage required for the record and the record offsets of the record's components. To illustrate this, consider the record type declaration in the example, i.e.,

```
rekord = record
     r1: integer;
     r2: array[-1..2] of char;
     r3: real;
   end;.
```

The corresponding information created by the definition in the symbol table is

identifier	id_type	element_type	arblk_ref	p_type	level	offset
rekord	type	record	3	0	1	10
r1	variable	integer	0	0	2	0
r2	variable	array	1*	1	2	1
r3	variable	real	0	1	2	9

The 'offset' field in the entry for the record identifier, i.e., 'rekord', indicates the total amount of storage required for the record. This value is used to decide the amount of storage to allocate when an identifier is declared with the record name as type, e.g., var recd: rekord. The 'offset' field of the entries for the record's components gives the record offsets of the components and are used when translating a record type variable. For example, consider the code fragment

```
root := recd.r3;
```

and its translation

LE22 -9 SE3.

The translation of the code fragment uses the record offset of r3, i.e., in the translation, 9. Finally, the 'arblk_ref' field of the entry for the record contains a value that is used as an index to the table containing information about the blocks in the program. The 'arblk_ref' field of the entries for the record's components will only be use if the component is of type array, e.g., r3, or another record. The use of this field, 'arblk_ref', will be explained in more detail in later sections.

9.4 Creation and Usage of Information from Variable Declarations

Aside from allocating storage space to variables during variable declaration, information must also be kept in order to access this storage space correctly. The declaration of a variable can proceed in several ways depending on the type of the variable. But, first we shall consider the declaration of standard type variables and the corresponding information created in the symbol table. Consider the variable declaration

```
int: integer;
root: real;
bool: boolean;
ch: char;
```

The information created in the symbol table by this declaration is

identifier	id_type	element_type	arblk_ref	p_type	level	offset
int	variable	integer	0	1	1	2
root	variable	real	0	1	1	3
bool	variable	boolean	0	1	1	4
ch	variable	character	0	1	1	5

Of importance during variable declaration are the values of the 'level' and 'offset' fields. The 'level' field is used in a comparison to decide whether the variable is local or non-local. The decision is made by comparing the value in the 'level' field with that of the textual level of the procedure where the identifier is used. If the difference is zero then it is a local access, otherwise it is a non-local access. The 'offset' field, on the other hand, indicates the offset from the local environment of the cell allocated to the variable. The translation of variables then proceeds by using the value of the 'offset' field as the raw operand of the instruction involving access to the variable. Consider for example the code fragment

```
int := 100;
```

assuming local access, the translation then is

```
L100 SE2.
```

The 2 in the instruction SE2 came from the 'offset' field of the entry for the identifier 'int'.

Next, is when variables are declared with array or record type. The most important fields during translation for array or record type identifiers are the 'offset' and the 'arblk_ref' fields. The 'offset' field gives the offset from the local environment of the first cell allocated to the data structure. This piece of information is, of course, important when translating array or record type variables. The second field that plays an important role in the translation of array and record type variables is the

'arblk_ref' field. The value of this field is used as an index to another data structure. These data structures, as mentioned earlier, will provide additional information about the array and record. Consider for example the variable declaration

```
table: array[0..2] of array[1..3] of integer;
recd: rekord;
arrecd: array[3..5] of rekord;
```

This declaration creates the following entries in the symbol table

identifier	id_type	element_type	arblk_ref	p_type	level	offset
table	variable	array	2*	1	1	6
recd	variable	record	3**	1	1	22
arrecd	variable	array	4*	1	1	32

The values with * are used as indices to a data structure where information about an array is stored and values with ** are used as indices to a data structure where information about a record or a block is stored.

For array type, the value of the 'arblk_ref' field is used as an index to a data structure which will provide information about the size, lower and upper bounds, and the dope vector of the array. The following is the table containing information about the arrays in the example declaration given in the beginning of this chapter.

arrays	element_type	arblk_ref	low	high	elm_size	total_size	dope_v_size-1
1	character	0	-1	2	1	8	3
2*	array	3*	0	2	3	16	6
3	integer	0	1	3	1	7	3
4*	record	3	3	5	10	37	6

So information about the variable 'table', can be known by using the value of its 'arblk_ref' field as an index to the table above. Further, note that the table of array information also contain an 'arblk_ref' field. The reason is that the elements of the array may also be of array or record type.

Additional information about the records defined in the declaration are provided by another data structure. This data structure contains information about the size of the record. To illustrate this, we present the table created for the example declaration given at the beginning of this chapter

blocks	parameter_size	variable_size
1	0	0
2	0	66
3""	0	10
4	4	2
5	2	0
6	0	0

9.5 Creation and Usage of Information from Procedure Declarations

With procedure declarations, we shall be interested in the heading part. The procedure heading consists of the procedure name and the formal parameters. Again, consider the procedure declaration in the example, i.e.,

```
pass( valuep: integer; var varp: integer;
      procedure procp(procedure inprocp) );
```

The entry in the symbol table created from the name of the procedure is

identifier	id_type	element_type	arblk_ref	p_type	level	offset
pass	procedure	no type	0	4*	1	4

The name of the procedure is important because the same name is used when calling the procedure. Consequently, the information stored in the symbol table for the procedure name should provide enough information to translate a procedure call. The 'level' field will be used when setting the static environment of the procedure. The 'offset' field contains the entry point of the procedure and the 'p_type' field is used as an index to the table containing the information about blocks, i.e.,

blocks	parameter_size	variable_size
1	0	0
2	0	66
3	0	10
4*	4	2
5	2	0
6	0	0

Note that from such information stored in the symbol table, assuming 'parm' is declared in the body of the procedure where it is called, the translation of a call

```
parm(int,int,p);
```

can be translated directly to (parameters are translated separately)

```
UO LEO C@4 D4.
```

The 4 in C@4 comes from the 'offset' field and the 4 in D4 is provided by the table containing the information about the block, of course through the use of the 'arblk_ref' field. The 0 in the instruction U0 is computed using the value in the 'level' field.

The next part to consider in a procedure declaration is the formal parameter part. The information created for formal parameters is similar to that for a variable declaration. This is obvious because parameters in PASCAL are actually considered as local variables. Now, we show the entries created by the parameters in the declaration of procedure 'parm'

identifier	id_type	element_type	arblk_ref	p_type	level	offset
valuep	variable	integer	0	1	2	-8
varp	variable	integer	0	0	2	-7
procp	procedure	no type	5	2	2	-6
inprocp	procedure	no type	6	2	3	-6

But unlike the entries for variables, where the 'p_type' field is ignored, in the declaration of the parameters the 'p_type' field has to be set correctly. The reason, of course, is that when parameters are used in the body of the procedure the translation will depend upon their type. Take for example, the following procedure declaration and translation of its statements.

```
procedure one(    a: integer;
                  var b:integer;
                  procedure p);
  begin
       a := 100;                          L100 SE-5
       b := 200;                          LIE-6 P L200 SH
       p;                                 LIE-7 CIE-8 DX8000
  end;
```

Note that the translation of each type of parameter is quite different from that of another.

CHAPTER 10

CODE IMPROVEMENT

There is always room for improvement of the code generated by a compiler. In a compiler consisting of a front end that translates to a common intermediate language and a back end that translates to a machine's assembly language, improvement of object code can be performed in three conceptual places [Tanenbaum, et. al., 1982].

The first place is to do the improvement in the front end. The decision to do it in the front end would consequently require that the translator be highly specialized. What we mean by a highly specialized translator is a translator that attempts to generate the best code that it can possibly generate for a particular source code fragment. Usually this type of translator is too complicated to construct and thus would require a high development effort. In addition, such translators will increase the compilation time of source programs because the compiler will have to carry out numerous tests to get better object code for a certain source code fragment. But no matter how specialized the translator is, it will still miss some possible improvements in the source code fragments translated separately by the translator. For example, the statements

```
a := b + c;
d := a + d;
```

will be translated by a highly specialized translator to

```
Lb +c Sa
La +d Sd.
```

Obviously, the translator fails to detect the possible improvement of the instructions Sa La to simply Sa. To catch these possible improvements, however, the compiler should do further improvement on the intermediate code. This brings us to the second conceptual place, i.e., doing the improvement on the intermediate code.

Since doing the improvement in the front end may still require another pass through the intermediate code to catch every possible improvement, it is usually advisable to do all the code improvement on the intermediate code and merely construct a simple front end translator. In this way, the development of the translator will not involve too much effort. Moreover, since the intermediate language does not change, the optimization procedures will be the same for all front ends or back ends.

The last conceptual place is to do the improvement in the back end. This possiblity seems to be the most profitable. The reason is that if the objective is to catch all possible improvement in the code, then improvement should be done in the code that is finally executed. But, this would mean that for every new back end, a new code improver must be written. Note that a possible improvement in one machine may not be possible in the other. For example,

```
move.l 4(a1), d0
move.l d0, 10(a0)
```

are Motorola 68000 instructions which can be improved to

```
move.l 4(a1), 10(a0).
```

But a similar set of instructions in another machine that does not allow

memory to memory copy, such code can not be improved at all.

Improvement is usually done on the intermediate code to avoid the greater development effort in doing the improvement in the front end and the back end. Although doing the improvement on the intermediate code will not catch all possible improvement, the difference compared to doing it in the back end is usually slight. This is because each intermediate code is usually mapped to the most efficient actual machine code.

There are several methods of improving intermediate code. But, the succeeding sections will concentrate only on one method, the peephole optimization technique.

10.1 Peephole Optimization

Peephole optimization can actually be used to improve intermediate and actual machine code. The method works by looking at a small range of instructions, at least two instructions, and replacing them by more efficient instructions. This small range of instructions is referred to as the peephole. The code in the peephole may be contiguous, e.g.,

Peephole	Replacement
SE2 LIE2	SE2

or not contiguous, e.g.,

Peephole	Replacement
LE2 P ... SH	... SE2

The nature of the technique is that the replacement code for a sequence of instructions can be used for further improvement. For example,

Sequence of Instruction	Peephole	Replacement
LIE2 P LIE3 +H	P LIE3	PLIE3
LIE2 PLIE3 +H	PLIE3 +H	+IE3
LIE2 +IE3	+IE3 ***	

The *** signifies the instruction which follows +H.

One of the aims of code improvement is to improve the code in a manner that the run-time improvement is greater than the overhead introduced by the improvement procedures at compile time. The next section will discuss how this objective is approached by showing several methods adopted to implement a peephole optimizer.

10.2 Implementation of a Peephole Optimizer

One implementation method was described in Davidson and Fraser's paper "The design and application of a retargetable peephole optimizer" [Davidson and Fraser, 1980]. Their method works by examining the pair of instructions in the peephole and replacing them, if possible, with one instruction which has the same action. In case the pair of instructions can not be reduced to one intruction, the first of the two instructions gets emitted. The new instructions in the peephole then are the second instruction of the previous peephole and the instruction immediately following the previous pair of instructions. For example, consider the pair of PDP-11 instructions

MOV @R3, R2

```
ADD #2, R3
```

which can be replaced by an equivalent one instruction

```
MOV (R3)+, R2.
```

Another implementation of a peephole optimizer was described in Tanenbaum's et. al. paper "Using Peephole Optimization on Intermediate Code" [Tanenbaum, et. al., 1982]. The method was used to improve the intermediate code EM. The method uses a pattern/replacement table. The table consists of a collection of lines, each line having a pattern part (peephole) and a replacement part. In contrast to Davidson and Fraser's approach, which uses a constant number of instructions in the peephole, the pattern part (peephole) vary in number of instructions. Their method works by simply constructing the patterns and replacements in advance and these are looked up in the table during compilation. To avoid missing new patterns created by the replacements, the method repeats the matching process until no more match is found. Examples of pattern and replacement lines are given below.

Pattern	Replacement	Comment
LOC A LOC B ADD	LOC (A + B)	Add constants A and B
LOC 2 MUL	LOC 1 SHL	Change multiplication to shift

Note that the length of the pattern (number of instructions) varies and the replacement is not necessarily smaller in length than the pattern. It may be the same length but the replacement is known to be executed faster than the pattern, e.g., the change from multiplication to shifting.

Next, is a method which was used to improve the intermediate SLIM

code generated by the translator described in Chapter 5.

The method used is exactly the same as the one employed by Davidson and Fraser, except that the number of instructions in the peephole is allowed to increase depending on the kind of source code the translator is translating.

The extension allowing more than two instructions in some code fragment is essential because the translator generates code which is impossible to improve with only two instructions in the peephole. As an example of this, consider the code fragment,

```
a - b
```

assuming 'a' and 'b' to be the first two local variable of a procedure, then the translation of the given code fragment is

```
LIE2 P LIE3 PLH -H.
```

Using only two instruction in the peephole, this can be improved to

```
LIE2 PLIE3 PLH -H.
```

But the subsequent translation can be improved to

```
LIE2 -IE3
```

if three instructions are used in the peephole.

Below is a summary of all the patterns and their corresponding replacements used in the optimizer described.

Pattern	Replacement
J@m @m:	@m:
J@m J@n	J@m
P Lm	PLm
P Um Ln	Um PLn
Mm Mn	M(m+n)
Mm R	R
Lm -	L-m (only when m is a constant)
+0, *+0	**
-0, *-0	**
*1, **1.0	**
/1, */1.0	**
SEm LIEm	SEm
LIEm SEm	**
R R	R
PLm PLH <op>H	<op>m
PLm <op>H	<op>m

The ** means that there is no replacement. In short, the pattern is deleted. The <op> stands for all dyadic SLIM operators. There are many other patterns in SLIM that can be improved but we showed here only those patterns which are actually generated by the translator described in Chapter 5.

The problem of determining the number of instructions in the peephole for a particular source code fragment can actually be decided by the manner the translator translates the code fragment. Take for example the same code fragment above, i.e.,

a - b,

and suppose that the translator translates the right operand first, this would mean that only two intructions in the peephole are enough to improve the code to its best possible form. To illustrate this point, consider the translation when the right operand is translated first. The translation will be

LIE3 P LIE2 -H

which can be improved to

LIE3 PLIE2 -H

and finally to

LIE3 -IE2.

It will be shown in appendix 5 that the code improver described introduces a negligible overhead to the compilation of source programs but improves the execution time by a reasonable amount.

CHAPTER 11

CONCLUSIONS

This project arose from a challenge by J. E. L. Peck to the author to use SLIM (which has been used only in implementing BCPL) as an intermediate language for PASCAL. A simple answer to this challenge is that SLIM could be a suitable vehicle for the implementation of PASCAL. But, a more interesting question is "How suitable is it?". Consequently, the question "How easy is it to generate SLIM?" could be asked. Section 10.1 will discuss some of the answers to these questions.

Using a new intermediate language in implementing PASCAL can only be justified if it proves to have at least the same run-time efficiency as the more popular intermediate language P-code. To check this, section 10.2 will present an execution time comparison of SLIM and the P-machine.

11.1 Suitability of SLIM as Target Language for PASCAL

Since SLIM was designed as a target language for translation of BCPL, the suitability of SLIM for PASCAL can be approximated by looking at the differences between the two languages and investigating whether these differences can be handled by SLIM. Obviously, the differences of concern will be those features supported in PASCAL but not in BCPL. It is a waste of time to look at the features common to both languages because SLIM was designed for them and therefore must be suitable.

The first difference between the two languages is the extent of the

environment of a procedure. The environment of a BCPL procedure apart from global and static variables is the data segment allocated to it on the stack. A PASCAL procedure environment on the other hand, is much wider because it supports non-local variables. But, as shown in Chapter 4, SLIM could be extended to be able to represent the environment of block-structured languages (like PASCAL) that support non-local variables. This extension is through the introduction of another cell in the SLIM link, the U register, and another instruction, the U instruction. This was an easy extension to make since it was carried out without affecting the design objectives of SLIM.

Another difference is with the standard data types available in the two languages. BCPL has only one standard data type, the bit pattern. PASCAL on the other hand, has four: integer, character, boolean, and real. The difference in standard data types supported by the two languages does not cause serious problems to the implementor because PASCAL variables and constituents of PASCAL expressions, where data type matters, are checked at compile time. This suggests that SLIM can be a suitable target language for languages that support several standard data types as long as checks of the data type are carried out at compile time. There is no easy way that SLIM can check the data type at run-time.

BCPL supports only one structured type, the vector (one-dimensional array). This brings us to another difference between the two languages. PASCAL supports not only one-dimensional array but also arrays with more than 1 dimension (multidimensional arrays). Chapter 6 shows that multidimensional arrays can be translated to SLIM using either of the two most common methods of setting up and accessing an element of an array. The

two methods are the indirect access via pre-calculated addresses method and multiplicative subscript calculation method. Therefore, SLIM can be used as a target language for languages that support multidimensional arrays.

Just like multidimensional arrays, record type structures are not really supported in BCPL (although there are primitive field selectors), but are supported in PASCAL. Chapter 6 shows that it is possible to use SLIM to set up a record structure and access it. SLIM can even be used to access record structures with more complicated components like arrays and even other records. Hence, SLIM can be a target language for languages that support record type structures.

The next difference between PASCAL and BCPL is the use of arrays (vectors) as parameters. Passing arrays (vectors) by value to a procedure is not supported in BCPL. BCPL allows only the passing of vectors by address. Passing arrays (and records) by value to a procedure requires the copying of the whole structure to the parameter area (in the stack) of the called procedure. Using the existing SLIM instructions, the process proves to be an expensive one. To illustrate this, consider the following example

PASCAL	SLIM
program pass(output);	
type vector = array[0..1] of char;	
var a: vector;	
procedure accept(b: vector);	$"accept" @3: D6 M1
begin	
end;	
begin	
accept(a)	LE2 SE1
	LIE1 LI0 P LIE1 LI1 P

```
                                        LIE1 LI2 P LIE1 LI3 P
                                        LIE1 LI4 P LIE1 LI5 P
                                        U0 LE0 C03 DX006
        end.
```

The number of times the address of the first cell allocated to the array is loaded in the accumulator seems to suggest that the translation can be improved.

One way of improving the code might be to introduce a multiple copy instruction in SLIM. The instruction that we mean is one whose operand is the number of cells to be copied to the top of stack and whose starting cell address is in the accumulator. The new instruction's action is similar to load and store subscripted cell (LI and SI) but it uses the top of stack as the source or destination instead of the accumulator.

Another way of improving the code might be to introduce a new register to the existing SLIM registers. The register we propose is similar to an address register of the MC68000. This means that whenever a load or store subscripted cell instruction is executed, the starting address is in this new address register instead of the accumulator. With the starting address in the new address register, the translation of

```
LIE1 LI0 P LIE1 LI1 P
LIE1 LI2 P LIE1 LI3 P
LIE1 LI4 P LIE1 LI5 P
```

can be improved to

```
LI0 P LI1 P LI2 P LI3 P LI4 P LI5 P.
```

Note that this would consequently require a new basic load and store instruction to set the value of and copy the value in this new register.

A similar situation to passing an array (or record) by value as a parameter happens when an array (or record) type variable is assigned to another array (or record) type variable. BCPL does the assignment of a vector to another vector by setting the value of the destination to a value equal to the address of the first cell allocated to the source vector. PASCAL, on the other hand, copies the contents of the source array (or record) to the storage allocated to the destination array (or record). Again, in this situation, code generation would have been simpler if there were a multiple copy instruction in SLIM.

Since situations like assigning a data structure to another and passing an array (or record) by value can be handled by the existing SLIM instructions (although ineffiently), we do not strongly propose that the possible extensions we mentioned above be incorporated in SLIM. The main reason is that, these situations are seldom used in PASCAL programs.

Finally, the incompatiblity on the handling of output in the two languages is another difference. BCPL will flush its output buffer only when a newline character is sent to it. PASCAL handles its output buffer differently. The output buffer is flush every time an output statement (write, writeln, etc) is complete. The difference can well be seen in the following programs

PASCAL	BCPL
program outbuffer(output);	LET start() BE

```
var n: integer;                    {1 LET n = ?
begin
  write(' Number Please: ');          writes(" Number Please: ")
  read(n)                             n := readn() }1
end.
```

The PASCAL program will flush the string ' Number Please: ' on the screen before executing the read statement. The corresponding BCPL program will expect an input but will output nothing on the screen.

The problem of output incompatibility, however, can be solved by flushing the output buffer everytime a 'write' statement is executed. Chapter 8 discusses this incompatibility thoroughly.

In conclusion, the existing SLIM instructions plus the extension to handle non-local variables are enough for block structured languages like PASCAL to be translated to SLIM.

Having answered the suitability of SLIM as a target language for languages like PASCAL, let us consider the question of how easy it is to generate SLIM code. The chapters dealing with the translation of PASCAL source fragments to SLIM code illustrate that such translation is as easy as translating PASCAL to P-code. The translator, which is strictly one-pass, shows that SLIM code can be generated in one pass like the more common intermediate language P-code.

Since generating SLIM code is as easy as generating P-code, the next obvious question is "How good is the quality of the SLIM code generated?". The following section will answer this question by comparing the execution time of SLIM code to its corresponding P-code.

11.2 Run-time Speed

Using SLIM as an intermediate code instead of P-code will make the compilation of programs a little longer. This is because SLIM is further translated to machine's assembly language (which of course is still part of the compilation process). P-code, on the other hand, is already the assembly language of a hypothetical stack-oriented interpreted machine (P-machine) and thus such overhead in the compilation process is nonexistent. But this compilation overhead is insignificant compared to the improvement of execution time when SLIM is used instead of P-code.

To illustrate this point, the following programs

1. Ammann's implementation of Knuth's algorithm on the computation of the date of Easter (see appendix 2).

2. Sorting of 1000 data items using the quicksort algorithm, i.e.,

```
program quicksort(output);
const n = 1000;
var i, z: integer;
       a: array[1..n] of integer;
procedure sort(left, right: integer);
  var i, j, x, w: integer;
  begin
     i := left;
     j := right;
     x := a[(i+j) div 2];
     repeat
        while a[i] < x do i := i + 1;
        while x < a[j] do j := j - 1;
```

```
            if i <= j then
               begin
                  w := a[i];
                  a[i] := a[j];
                  a[j] := w;
                  i := i + 1;
                  j := j - 1;
               end
         until i > j;
         if left <  j then sort(left, j);
         if left < right then sort(i, right);
      end;

   { generate random sequence of numbers }
   begin
      z := 1729;
      for i := 1 to n do
         begin
            z := (131071 * z) mod 2147483647;
            a[i] := z
         end;
      sort(1,n)
   end.
```

3. Wirth's implementation of the eight queens problem (see [Wirth, 1977]).

4. Multiplying a 20 by 20 matrix, i.e.,

```
   program matrixmult(output);
   const n = 20;
   var i, j, k: integer;
                x: real;
          m, r: array[1..n,1..n] of real;
   begin
      for i := 1 to n do
          for j := 1 to n do m[i, j] := 1.0;
```

```
        for i := 1 to n do
            for j := 1 to n do
               begin
                  x := 0;
                  for k := 1 to n do x := m[i, k] * m[k, j] + x;
                  r[i, j] := x;
               end
    end.
```

were translated to SLIM and to P-code. The resulting code was then executed. The system used for translation to P-code and execution of it is the Berkeley's PASCAL compiler/interpreter system. The results are summarized below.

Execution Time (in seconds)

Program	Runtime System	
	SLIM	P-Machine
Date of Easter	0.38	2.34
Quicksort	1.36	6.44
Eight Queens	3.16	11.41
Matrix Multiplication	0.96	4.16

The summary of execution times shows that SLIM code is executed significantly faster than P-code. This can be attributed to the fact that P-code is interpreted while SLIM code is translated to machine language and therefore directly executed. The difference between the execution times might not be that much if P-code is macroexpanded to its machine language equivalents. But, we should remember that P-code was not designed to be macroexpanded; thus compilation time may increase significantly if this approach of executing P-code is taken.

REFERENCES

1. Aho, A. V., and Ullman, J. D. The Principles of Compiler Design, Addison-Wesley Publishing Co., New York, N.Y., 1977.

2. Ammann, U. "On code generation in a PASCAL compiler", Software - Practice and Experience, 7, No. 3, 391-423 (1977).

3. Berry, R. E. "Experience with the PASCAL P-Compiler", Software - Practice and Experience, 8, No. 5, 617-627 (1978).

4. Bornat, R. Understanding and Writing Compilers, MacMillan Press, London, 1979.

5. Bron, C. and De Vries, W. "A PASCAL Compiler for PDP-11 Minicomputers", Software - Practice and Experience, 6, No. 1, 109-116 (1976).

6. Colemann, S. S., Poole, P. C., and Waite, W. M. "The mobile programming system: Janus", Software - Practice and Experience, 4, No. 1, 5-23 (1974).

7. Davidson, J. W., and Fraser, C. W. "The design and application of a retargetable peephole optimizer", ACM Transactions on Programming Languages and Systems, 2, No. 2, 191-202 (1980).

8. Fox, M. Machine Architecture and the Programming Language BCPL, University of British Columbia MSc Thesis, Vancouver, Canada, 1978.

9. Grosse-Lindemann, C. O., and Nagel, H. H. "Postlude to a PASCAL-Compiler bootstrap on a DECSystem-10", Software - Practice and Experience, 6, No. 1, 29-42 (1976).

10. Jensen, K., and Wirth, N. PASCAL User Manual and Report, Springer Verlag, Berlin-Heidelberg-New york, 1974.

11. Knuth, D. E. The Art of Computer Programming, Volume 1, Addison-Wesley, New York, 1968.

12. Lecarme, O. "Structured programming, programming teaching and the language Pascal", Sigplan Notices, 9, No. 7, 15-21 (1974).

13. Lecarme, O. "Pascal and portability", Third Annual Computer Studies Symp.: Pascal - the language and its implementations, Southhampton (1977).

14. Lecarme, O., and Peyrolle-Thomas, M. C. "Self-compiling Compilers: An Appraisal of their Implementation and Portability", Software - Practice and Exprerience, 8, No. 2, 149-170 (1978).

15. McKeenan, W. M. "Peephole Optimization", Comm. ACM, 8, No.7, 443-444 (1965).

16. Nori, K. V., Ammann, U., Jensen, K., and Nageli, H. The Pascal(P) Compiler Implementation Notes, Institut fur Informatik, Eidgenossische Technische Hochschule, Zurich, 1975.

17. Parsons, A. L. " A Microcomputer Pascal Cross Compiler",

Proceedings of Spring Compcon 78, San Francisco, February-March, 1978, IEEE, 145-150 (1978).

18. Peck, J. E. L. The Essence of Portable Programming, In preparation.

19. Randell, B., and Russell, L. J. Algol 60 Implementation, Academic Press, London and New York, 1964.

20. Richards, M. "The portability of the BCPL compiler", Software - Practice and Experience, 1, No. 2, 135-146 (1971).

21. Russell, D. L., and Sue, J. Y. "Implementation of a Pascal compiler for the IBM 360", Software - Practice and Experience, 6, No. 3, 371-376 (1976).

22. Tanenbaum, A. S., and van Staveren, H., and Stevenson, J. W. "Using Peephole Optimization on Intermediate Code", ACM Transactions on Programming Languages and Systems, 4, No. 1, 21-36 (1982).

23. Welsh, J., and Quinn, C., "A Pascal compiler for the ICL 1900 series computer", Software - Practice and Experience, 2, 1, 73-77 (1972).

24. Welsh, J. "Two ICL 1900 PASCAL compilers", Third Annual Computer Studies Symposium: Pascal - the Language and its Implementations, Southampton (1977).

25. Wichmann, B. A., and Sales, A. H. J. "A Pascal Validation Suite", Unpublished, 1979.

26. Wirth, N. "The Design of a PASCAL Compiler", Software - Practice and Experience, 1, No. 3, 303-333 (1971).

27. Wirth, N. Algorithm + Data Structures = Programs, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1976.

28. Wulf, W., Johnson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M. The Design of an Optimizing Compiler, American Elvesier, New York, N. Y., 1975.

APPENDIX 1

Example 1: Preorder Tree Traversal

The following program solves the problem of building a tree and traversing it in preorder. In addition, it prints the value of the nodes of the tree that it visits.

```
program preorder(input, output);
const maxlength = 20;
type  node = record
        value: char;
        leftlink, rightlink: integer;
      end;
var   tree: array[1..maxlength] of node;
      stack: array[1..maxlength] of integer;
      top, i: integer;
function empty: boolean;
   begin
      if top = 0
         then empty := true
         else empty := false;
   end;
procedure push(newtop: integer);
   begin
      if top >= maxlength
         then writeln(' Error in PUSH: Stack Overflow ');
         else
            begin
               top := top + 1;
               stack[top] := newtop;
            end;
   end;
procedure pop(var topvalue: integer);
   begin
      if empty
```

```
            then writeln(' Error in POP: Stack is Empty ');
            else
               begin
                  topvalue := stack[top];
                  top := top - 1;
               end;
      end;
   procedure preprint(root: integer);
      var thisnode: integer;
      begin
         top := 0;
         push(root);
         repeat
            pop(thisnode);
            writeln(tree[thisnode].value);
            if tree[thisnode].rightlink <> 0
               then push(tree[thisnode].rightlink);
            if tree[thisnode].leftlink <> 0
               then push(tree[thisnode].leftlink);
         until empty
      end;
   begin
      i := 0;
      repeat
         i := i + 1;
         readln(tree[i].value,
                tree[i].leftlink,
                tree[i].rightlink);
      until eof(input);
      preprint(1);
   end.
```

The choice of the above program was due to several reasons. One is that it illustrates the declaration of and access (local and non-local) to variables including arrays and records. Further, the program contains procedures with value and variable parameters, conditional statements, and repetitive statements.

The complete translation of the program is shown below.

PASCAL	Unimproved SLIM
`program preorder(input, output);`	`$$"preorder" J@2 @1: D0 M1`
`const maxlength = 20;`	
`type node = record`	
`value: char;`	
`leftlink, rightlink: integer;`	
`end;`	
`var tree: array[1..maxlength] of node;`	`L-64 PL20 PL1 PL20 P`
	`M1 M1 M1 M1 M1 M1 M1 M1 M1 M1`
	`M1 M1 M1 M1 M1 M1 M1 M1 M1 M1`
	`M1 M1 M1 M1 M1 M1 M1 M1 M1 M1`
	`M1 M1 M1 M1 M1 M1 M1 M1 M1 M1`
	`M1 M1 M1 M1 M1 M1 M1 M1 M1 M1`
	`M1 M1 M1 M1 M1 M1 M1 M1 M1 M1`
`stack: array[1..maxlength] of integer;`	`L-24 PL20 PL1 PL20 M20`
`top, i: integer;`	`M1 M1`
	`J@3`
`function empty: boolean;`	`$"empty" @4: D0 M1 M1`
	`J@5 @5:`
`begin`	
`if top = 0`	`U1 LIE90 P L0 =H F@6`
`then empty := true`	`L-1 SE2 J@7`
`else empty := false;`	`@6: L0 SE2`
`end;`	`@7: LIE2 R $: $`
`procedure push(newtop: integer);`	`$"push" @8: D1 M1`
	`J@9 @9:`
`begin`	
`if top >= maxlength`	`U1 LIE90 P L20 PLH >=H F@10`
`then writeln(' Error in PUSH:`	
`Stack Overflow ');`	`L@11 C1065 D1 C1067 D0 J@12`
`else`	`@10:`
`begin`	
`top := top + 1;`	`U1 LIE90 P L1 +H U1 SE90`
`stack[top] := newtop;`	`U1 LE66 SE1 P L10 +H PL0`

```
        end;
  end;


procedure pop(var topvalue: integer);

  begin
    if empty
      then writeln(' Error in POP:
                     Stack is Empty ');
      else
        begin
          topvalue := stack[top];

          top := top - 1;
        end;
  end;


procedure preprint(root: integer);
  var thisnode: integer;

  begin
    top := 0;
    push(root);
    repeat
      pop(thisnode);
      writeln(tree[thisnode].value);

      if tree[thisnode].rightlink <> 0

        then push(tree[thisnode].rightlink);

      if tree[thisnode].leftlink <> 0
```

```
    P U1 LIE90 +H +H P LIE-5 SH
@12:
R  $:
@11: D" Error in PUSH:
        Stack Overflow  "  $
$"pop" @13: D1 M1
J@14  @14:

LIE-4 C@4 DX8000 F@15

L@16 CIG65 D1 CIG67 D0 J@17
@15:

U1 LE66 SE1 P LI0 +H PL0 P U1
  LIE90 +H +H LI0 P LIE-5 SI0
U1 LIE90 P L1 PLH -H U1 SE90
@17
R  $:
@16: D" Error in POP:
        Stack is Empty "  $
$"preprint"  @18: D1 M1
M1
J@19 @19:

L0 U1 SE90
LIE-5 P LIE-4 C@8 DX8001
@20:
   LE2 P LIE-4 C@13 DX8001
   U1 LE2 SE1 P LI0 +H PL0
      P LIE2 +H *3 +H LI0
      CIG60 D1 CIG67 D0
   U1 LE2 SE1 P LI0 +H PL0
      P LIE2 +H *3 +H -2 LI0
      P L0 ~=H F@21
    U1 LE2 SE1 P LI0 +H PL0
         P LIE2 +H *3 +H -2 LI0
         P LIE-4 C@8 DX8001
   @21:
   U1 LE2 SE1 P LI0 +H PL0
```

```
                                                    P LIE2 +H *3 +H -1 LIO
                                                    P LO ~=H F@22
        then push(tree[thisnode].leftlink);      U1 LE2 SE1 P LIO +H PLO
                                                    P LIE2 +H *3 +H -1 LIO
                                                    P LIE-4 C@8 DX8001
                                                 @22:
   until empty                                   LIE-4 C@4 DX8000 F@20
 end;                                            R  $:  $
                                                 @3:
begin
  i := 0;                                        LO SE@1
  repeat                                         @23:
    i := i + 1;                                    LIE@1 P L1 +H SE@1
    readln(tree[i].value,                          LE2 SE1 P LIO  +H PLO
                                                    P LIE@1 +H *3 +H P
                                                    L3 CIG84 D1 SH
           tree[i].leftlink,                       LE2 SE1 P LIO  +H PLO
                                                    P LIE@1 +H *3 +H -1 P
                                                    L1 CIG84 D1 SH
           tree[i].rightlink);                     LE2 SE1 P LIO  +H PLO
                                                    P LIE@1 +H *3 +H -2 P
                                                    L1 CIG84 D1 SH

                                                   L5 CIG84 D1
  until eof(input);                              CIG8@ D0 F@23
  preprint(1);                                   L1 P U0 LE0 C@18 DX8001
end.                                             R $:  $
                                                 @2: L@1  SG1 .
```

The SLIM code opposite the PASCAL source above is the initial translation generated by the translator. This code passes through a code improver before it is finally emitted. The following is the code produced by the code improver. The improvements are shown in bold letters.

PASCAL	Improved SLIM
`program preorder(input, output);`	`$$"preorder" J@2 @1: D0 M1`

```
const maxlength = 20;
type node = record
       value: char;
       leftlink, rightlink: integer;
      end;
var tree: array[1..maxlength] of node;
   stack: array[1..maxlength] of integer;
  top, i: integer;

function empty: boolean;

   begin
     if top = 0
       then empty := true
       else empty := false;
   end;
procedure push(newtop: integer);

  begin
    if top >= maxlength
      then writeln(' Error in PUSH:
                     Stack Overflow ');
      else
        begin
          top := top + 1;
          stack[top] := newtop;

        end;
  end;


procedure pop(var topvalue: integer);

  begin
    if empty
      then writeln(' Error in POP:
                     Stack is Empty ');
        else
          begin
```

```
L-64 PL20 PL1 PL20 P M60
L-24 PL20 PL1 PL20 M22

J@3
$"empty" @4: D0 M2
@5:

U1 LIE90 =0 F@6
   L-1 SE2 J@7
@6: L0 SE2
@7: LIE2  R  $:  $
$"push"  @8: D1 M1
@9:

U1 LIE90 >=20 F@10

L@11 CIG65 D1 CIG67 D0 J@12
@10:

  U1 LIE90 +1 U1 SE90
  U1 LE66 SE1 P LI0 +H PL0
    U1 +IE90 +H PLIE-5 SH
@12:
R  $:
@11: D" Error in PUSH:
         Stack Overflow  "  $
$"pop" @13: D1 M1
@14:

LIE-4 C@4 DX8000 F@15

L@16 CIG65 D1 CIG67 D0 J@17
@15:
```

```
          topvalue := stack[top];

          top := top - 1;
        end;
  end;


procedure preprint(root: integer);
  var thisnode: integer;

  begin
    top := 0;
    push(root);
    repeat
      pop(thisnode);
      writeln(tree[thisnode].value);


      if tree[thisnode].rightlink <> 0

        then push(tree[thisnode].rightlink);


      if tree[thisnode].leftlink <> 0

        then push(tree[thisnode].leftlink);


    until empty
  end;

begin
  i := 0;
  repeat
    i := i + 1;
```

```
U1 LE66 SE1 P L10 +H PL0
  U1 +1E90 +H L10 PL1E-5 S10
U1 L1E90 -1 U1 SE90
@17
R  $:
@16: D" Error in POP:
        Stack is Empty "  $
$"preprint"  @18: D1 M2

@19:

L0 U1 SE90
L1E-5 PL1E-4 C@8 DX8001
@20:
   LE2 PL1E-4 C@13 DX8001
   U1 LE2 SE1 P L10 +H PL0
      +1E2 *3 +H L10
      C1G60 D1 C1G67 D0
   U1 LE2 SE1 P L10 +H PL0
      +1E2 *3 +H -2 L10
      "=0 F@21
    U1 LE2 SE1 P L10 +H PL0
         +1E2 *3 +H -2 L10
         PL1E-4 C@8 DX8001
   @21:
   U1 LE2 SE1 P L10 +H PL0
      +1E2 *3 +H -1 L10
      "=0 F@22
  U1 LE2 SE1 P L10 +H PL0
         +1E2 *3 +H -1 L10
         PL1E-4 C@8 DX8001
   @22:
L1E-4 C@4 DX8000 F@20
R  $:  $
@3:

L0 SE91
@23:
   L1E91 +1 SE91
```

```
    readln(tree[i].value,
           tree[i].leftlink,
           tree[i].rightlink);

  until eof(input);
  preprint(1);
end.
```

```
LE2 SE1 P L10 +H PL0
   +IE01 *3 +H
   PL3 CIG84 D1 SH
LE2 SE1 P L10  +H PL0
   +IE01 *3 +H -1
   PL1 CIG84 D1 SH
LE2 SE1 P L10  +H PL0
   +IE01 *3 +H -2
   PL1 CIG84 D1 SH

  L5 CIG84 D1
CIG89 D0 F@23
L1 U0 PLE0 C@18 DX8001
R $:  $
@2: L@1  SG1 .
```

APPENDIX 2

Example 2: Date of Easter

Ammann [Ammann, 1977] gives an implementation of the algorithm of the computation of the date of Easter described in Knuth's "The Art of Computer Programming." The same implementation is given below, but with simpler declarations.

```
program easter(output);
const year1 = 1985;
      yearn = 3000;
var   year, month, day: integer;
procedure dateofeaster(y: integer; var d, m: integer);
   var g, c, x, z, b, e: integer;
   begin
      g := y mod 19 + 1;
      c := y div 100 + 1;
      x := 3 * c div 4 - 12;
      b := 5 * y div 4 - x - 10;
      e := (11 * g + 20 + z - x) mod 30;
      if e < 0 then e := e + 30;
      if (e = 25) and (g > 11) or (e = 24) then e := e + 1;
      d := 44 - e;
      if d < 21 then d := d + 30;
      d := d + 7 - (b + d) mod 7;
      if d > 31 then
            begin
               d := d - 31;
               m := 4;
            end
         else  m := 3;
   end;
begin
   for year := year1 to yearn do
      dateofeaster(year, day, month);
end.
```

The program was specifically chosen to illustrate the translation of expressions. Another reason is that the program clearly shows the difference between unimproved and improved SLIM code generated by the translator. Again, below is the unimproved translation opposite its equivalent PASCAL source code fragment.

PASCAL	Unimproved SLIM
`program easter(output);`	`$$"easter" J@2 @1: D0 M1`
`const year1 = 1985;`	
`yearn = 3000;`	
`var year, month, day: integer;`	`M1 M1 M1`
`procedure dateofeaster(`	`$"dateofester" @4: D3 M1`
`y: integer;`	
`var d, m: integer);`	
`var g, c, x, z, b, e: integer;`	`M1 M1 M1 M1 M1 M1`
	`J@5 @5:`
`begin`	
`g := y mod 19 + 1;`	`LIE-7 P L19 PLH /*H P L1 +H SE2`
`c := y div 100 + 1;`	`LIE-7 P L100 PLH /H P L1 +H SE3`
`x := 3 * c div 4 - 12;`	`L3 P LIE3 *H P L4 PLH /H P L12 PLH -H SE4`
`z := (8 * c + 5) div 25 - 5;`	`L8 P LIE3 *H P L5 +H P L25 PLH /H P L5 PLH -H SE5`
`b := 5 * y div 4 - x - 10;`	`L5 P LIE-7 *H P L4 PLH /H P LIE4 PLH -H P L10 PLH -H SE6`
`e := (11 * g + 20 + z - x) mod 30;`	`L11 P LIE2 *H P L20 +H P LIE5 +H P LIE4 PLH -H P L30 PLH /*H SE7`
`if e < 0 then`	`LIE7 P L0 PLH <H F@6`
`e := e + 30;`	`LIE7 P L30 +H SE7`
	`@6:`
`if (e = 25) and (g > 11) or (e = 24)`	`LIE7 P L25 =H F@7 P LIE2 P L11 PLH >H /\H T@7 P LIE7 P L24 =H \/H @7: F@8`
`then e := e + 1;`	`LIE7 P L1 +H SE7`
	`@8:`

```
    d := 44 - e;                          L44 P LIE7 PLH -H PLIE-6 S!0
    if d < 21                             LIE-6 L!0 P L21 PLH <H F@9
      then d := d + 30;                     LIE-6 L!0 P L30 +H PLIE-6 S!0
                                          @9:
    d := d + 7 - (b + d) mod 7;           LIE-6 L!0 P L7 +H P LIE6 P LIE-6
                                            L!0 +H P L7 PLH /*H PLH -H
                                            PLIE-6 S!0
    if d > 31 then                        LIE-6 L!0 P L31 PLH >H F@10
      begin
        d := d - 31;                        LIE-6 L!0 P L31 PLH -H PLIE-6
                                              S!0
        m := 4;                             L4 PLIE-5 S!0
      end                                 J@11
    else                                  @10:
      m := 3;                               L3 PLIE-5 S!0
                                          @11:
  end;                                    R  $:  $
begin
  for year := year1 to yearn do           @3: L1985 SE2 L3000 J@12
    dateofeaster(year, day, month);           @13: LIE2 P LE4 P LE3 P
                                                   U0 LE0 C@4 DX8003
                                              LIE2 +1 SE2 LH
                                              @12: P >=IE2 T@13 M-1
end.                                      R  $:  $
                                          @2: L@1 SG1 .
```

The improved equivalent (improvement in bold letters) of the above translation is

PASCAL	Improved SLIM
`program easter(output);`	`$$"easter" J@2 @1: D0`
`const year1 = 1985;`	
`yearn = 3000;`	
`var year, month, day: integer;`	**M4**
`procedure dateofeaster(`	`$"dateofester" @4: D3`
`y: integer;`	
`var d, m: integer);`	
`var g, c, x, z, b, e: integer;`	**M7**

```
  begin
    g := y mod 19 + 1;
    c := y div 100 + 1;
    x := 3 * c div 4 - 12;
    z := (8 * c + 5) div 25 - 5;
    b := 5 * y div 4 - x - 10;
    e := (11 * g + 20 + z - x) mod 30;

    if e < 0
      then e := e + 30;

    if (e = 25) and (g > 11) or (e = 24)

      then e := e + 1;

    d := 44 - e;
    if d < 21
      then d := d + 30;

    d := d + 7 - (b + d) mod 7;

    if d > 31
      then
        begin
          d := d - 31;
          m := 4;
        end
      else
        m := 3;

  end;
begin
  for year := year1 to yearn do
    dateofeaster(year, day, month);



end.
```

```
@5:

LIE-7 /*19 +1 SE2
LIE-7 /100 +1 SE3
L3 *IE3 /4 -12 SE4
L8 *IE3 +5 /25 -5 SE5
L5 *IE-7 /4 -IE4 -10 SE6
L11 *IE2 +20 +IE5 -IE4 /*30
  SE7
<0  F@6
    LIE7 +30 SE7
@6:
  LIE7 =25 F@7 PLIE2  >11 /\H
    T@7 PLIE7  =24 \/H  @7: F@8
    LIE7 +1 SE7
@8:
L44 -IE7 PLIE-6 S!0
LIE-6 L!0 <21 F@9
    LIE-6 L!0 +30 PLIE-6 S!0
@9:
LIE-6 L!0 +7 PLIE6 PLIE-6 L!0
  +H /*7 PLH -H PLIE-6 S!0
LIE-6 L!0 >31 F@10


    LIE-6 L!0 -31 PLIE-6 S!0
    L4 PLIE-5 S!0
J@11
@10:
    L3 PLIE-5 S!0
@11:
R  $:  $

@3: L1985 SE2 L3000 J@12
    @13: LIE2 PLE4 PLE3
        U0 PLE0 C@4 DX8003
    LIE2 +1 SE2 LH
    @12: P  >=IE2 T@13
R  $:  $
@2: L@1 SG1 .
```

Note the significant improvement of the code after it has gone through the code improver. The number of SLIM instructions is reduced from 235 instructions to 131 instructions. A significant improvement of more than fifty percent (>50%).

APPENDIX 3

Example 3: Towers of Hanoi

This next example is the recursive solution to the Towers of Hanoi problem.

```
program hanoi(input, output);
var maxring: integer;
      tower: array[1..3] of char;
procedure move(maxring: integer; a, b: integer);
   var c: integer;
   begin
      if maxring < 2
         then writeln(' Move ring  1 from tower ',
                                   tower[a],
                                   ' to tower ',
                                   tower[b])
         else
            begin
               c := 6 - a - b;
               move(maxring - 1, a, c);
               writeln(' Move ring ',
                       maxring:2,
                       ' from tower ',
                       tower[a],
                       ' to tower ',
                       tower[b]);
               move(maxring - 1, c, b);
            end
   end;
begin
   tower[1] := 'A';
   tower[2] := 'B';
   tower[3] := 'C';
   write(' Enter maximum rings please: ');
   readln(maxring);
```

```
    if maxring > 0 then move(maxring, 1, 2)
end.
```

The program was primarily chosen to illustrate how translation of programs involving recursion can be carried out. The translation given opposite the program below has already passed through the code improver.

PASCAL	SLIM
`program hanoi(input, output);`	`$$"hanoi" J@2 @1: D0`
`var maxring: integer;`	`M2`
`  tower: array[1..3] of char;`	`L-7 PL3 PL1 PL3 P M3`
	`J@3`
`procedure move(maxring: integer;`	`$"move" @4: D3`
`  a, b: integer);`	
`  var c: integer;`	`M2`
	`@5:`
`  begin`	
`    if maxring < 2`	`LIE-7 <2 F@6`
`      then writeln(' Move ring 1`	
`  from tower ',`	`L@7 CIG65 D1`
`  tower[a],`	`U1 LE3 SE1 P LI0 +H PL0 +IE-6 +H LI0 CIG60 D1`
`  ' to tower ',`	`L@8 CIG65 D1`
`  tower[b])`	`U1 LE3 SE1 P LI0 +H PL0 +IE-5 +H LI0 CIG60 D1`
	`CIG67 D0`
	`J@9`
`      else`	`@6:`
`        begin`	
`          c := 6 - a - b;`	`L6 -IE-6 -IE-5 SE2`
`          move(maxring - 1, a, c);`	**`LIE-7 -1 PLIE-6 PLIE2 PLIE-4 C@4 DX8003`**
`          writeln(' Move ring ',`	`L@10 CIG65 D1`
`  maxring:2,`	`LIE-7 PL2 CIG71 D2`
`  ' from tower ',`	`L@11 CIG65 D1`
`  tower[a],`	`U1 LE3 SE1 P LI0 +H PL0 +IE-6 +H LI0 CIG60 D1`

```
                    ' to tower ',
                    tower[b])

          move(maxring - 1, c, b);

      end

  end;
```

```
@9:
R $:
@7: D" Move ring 1 from tower "
@8: D" to tower "
@10: D" Move ring "
@11: D" from tower "
@12: D" to tower " $
@3:
```

```
begin
  tower[1] := 'A';
  tower[2] := 'B';
  tower[3] := 'C';
  write(' Enter maximum
         rings please: ');
  readln(maxring);
  if maxring > 0
    then move(maxring, 1, 2)
end.
```

```
LE3 SE1 P L!0 +H PL0 +1 +H PL'A SH
LE3 SE1 P L!0 +H PL0 +2 +H PL'B SH
LE3 SE1 P L!0 +H PL0 +3 +H PL'C SH

L@13 CIG65 D1 CIG5 D0
L1 CIG84 D1 SE2 L5 CIG84 D1
LIE2 >0 F@14
  LIE2 PL1 PL2 U0 PLE0 C@4 DX8003
@14: R  $:
@13: D" Enter maximum
        rings please: "  $$
@2: L@1 SG1 .
```

```
                    L@12 CIG65 D1
                    U1 LE3 SE1 P L!0 +H PL0 +1E-5 +H
                      L!0 CIG60 D1
                    CIG67 D0
          move(maxring - 1, c, b);  LIE-7 -1 PLIE2 PLIE-5 PLIE-4
                                      C@4 DX8003
```

The SLIM code in bold letters are the recursive calls made by the program.

APPENDIX 4

Example 4: Environment of Procedure Parameters

This last example is a program that can be used to test whether or not your PASCAL run-time system sets correctly the environment of procedures passed as parameters. It will print a message if the environment is set correctly, otherwise no message is printed. The program was based on one of the programs in Wichmann and Sales' PASCAL Validation Suite [Wichmann and Sales, 1979].

```
program environment(output);
var x, y: integer;
procedure p(procedure f(procedure a; procedure b);
            procedure g);
   var z: integer;
   procedure s;
      begin
         if (x = 2) and (y = 2) and (z =2) then
            writeln(' Yes! Environtment correctly set. ");
         x := x + 1;
      end;
   begin
      y := y + 1;
      z := y;
      if y = 1
         then p(f,s)
         else  f(g,s);
   end;
procedure q(procedure f; procedure g);
   begin
      f;
      g;
   end;
procedure r;
   begin
```

```
      end;
    begin
      x := 1;
      y := 0;
      p(q,r);
    end.
```

The example was chosen to illustrate the translation of procedures as parameters. The above program almost covers all possible calls to a procedure that is a parameter. Again, the translation is already an improved one.

PASCAL	SLIM
`program environment(output);`	`$$"environment" J@2 @1: D0`
`var x, y: integer;`	`M3`
	`J@3`
`procedure p(procedure f(procedure a;`	`$"p" @4: D4`
`procedure b);`	
`procedure g);`	
`var z: integer;`	`M2`
	`J@5`
`procedure s;`	`$"s" @6: D0 M1`
	`@7:`
`begin`	
`if (x = 2) and (y = 2) and (z =2)`	`U2 LIE2 =2 F@8 U2 PLIE3 =2`
	`/\H F@8 U1 PLIE2 =2 /\H @8: F@9`
`then writeln(' Yes!`	
`Environtment correctly set. ");`	`L@10 CIG65 D1 CIG67 D0`
	`@9:`
`x := x + 1;`	`U2 LIE2 +1 U2 SE2`
`end;`	`R $:`
	`@10: D" Yes! Environment`
	`correctly set. " $`
	`@5:`
`begin`	
`y := y + 1;`	`U1 LIE3 +1 U1 SE3`
`z := y;`	`U1 LIE3 SE2`

```
     if y = 1                                  U1 LIE3 =1 F@11
       then p(f,s)                             LIE-8 PLIE-7 PLI@12 U0 PLE0
                                                 PLIE-4 C@4 DX8004 J@13
       else  f(g,s);                           @11: LIE-6 PLIE-5 PLI@12 U0 PLE0
                                                 PLIE-7 CIE-8 DX8004
                                               @13:
   end;                                        R $:
                                               @12: D@6  $
procedure q(procedure f; procedure g);         $"q" @14: D4 P
                                               @15:
  begin
      f;                                       LIE-7 CIE-8 DX8000
      g;                                       LIE-5 CIE-6 DX8000
   end;                                        R  $:  $
procedure r;                                   $"r" @16: D0 P
                                               @17:
   begin
   end;                                        R  $:  $
                                               @3:
begin
   x := 1;                                     L1 SE2
   y := 0;                                     L0 SE3
   p(q,r);                                     LI@18 U0 PLE0 PLI@19 U0 PLE0
                                                 U0 PLE0 C@4 DX8004
end.                                           R  $:
                                               @18: D@14
                                               @19: D@16  $$
                                               @2: L@1 SG1  .
```

APPENDIX 5

Compilation and Execution Times Comparison of Improved and Unimproved SLIM Code

The translator (described in Chapter 5) was used to compile several programs to investigate the effect of the code improver (described in Chapter 10) on the compilation and execution times of source programs. Of course, attempting to improve the code will almost certainly degrade the compilation of source programs. But, if the improvement will decrease the execution time by at least the same amount as the increase in compilation time then the improvement carried out on the code is certainly worthwhile.

To see whether the method used by the code improver results in a positive effects, i.e., the decrease in execution time is greater than the increase in compilation time, the translator was used to translate the following programs:

1. Ammann's implementation of Knuth's algorithm on the computation of the date of Easter (see appendix 2).

2. Sorting of 1000 data items using the quicksort algorithm (see Chapter 11).

3. Wirth's implementation of the eight queens problem (see [Wirth, 1977]).

4. Multiplying a 20 by 20 matrix (see Chapter 11).

The programs were translated (with and without the code improver) and executed. The summary of compilation and execution times are given below.

Compilation Time (in seconds)

Program	With the Code Improver	Without the Code Improver
Date of Easter	0.46	0.41
Quicksort	0.71	0.67
Eight Queens	0.61	0.61
Matrix Multiplication	0.33	0.33

Execution Time (in seconds)

Program	Improved SLIM Code	Unimproved SLIM Code
Date of Easter	0.38	0.69
Quicksort	1.36	1.78
Eight Queens	3.16	4.07
Matrix Multiplication	0.96	1.11

It is clear from the summary of compilation and execution times that the degradation in compilation due to the introduction of the code improver is less than 0.1 second. But, the improvement in execution time is much more than 0.1 second. This shows that the introduction of the code improver is certainly worthwhile.

APPENDIX 6

SLIM Code and PASCAL-S P-code Comparison

The main difference between SLIM code and PASCAL-S P-code is in the manner the code is executed.

The PASCAL-S P-code was designed for interpretation. At run-time, i.e., during interpretation, the execution of some of the P-code instructions is dependent on the information stored in the symbol table. This means that the symbol table, created during compilation, must be accessible to the interpreter.

SLIM code, on the other hand, is further translated to the assembly language of the machine where the program is to run and is therefore directly executed. Each SLIM code instruction, once generated, can be translated to assembly language without extra information from the symbol table.

The dependence of PASCAL-S P-code on the symbol table suggests that SLIM code is lower in level compared to P-code.

The second difference between the two intermediate codes is the way stack segment, i.e., storage for parameters, stack linkage, local variables and working storage, is allocated to the called procedure.

In PASCAL-S, allocation of a stack segment is done before the control of execution is transfered to the called procedure. This means that

each procedure starts with the space for the local variables already allocated. To illustrate this point, consider the translation to P-code of the call to procedure "pass(i,i)" in the program given later in this appendix.

```
   PASCAL          P-code          Comment

pass(i, i)      MARKSTK  44       Allocate storage to local variables
                LOADVAL   1,  5   Pass the parameters
                LOADADR   1,  5
                CALL_FNP  6       Set up stack linkage and
                                  transfer control to the procedure
```

The P-code instruction "MARKSTK" allocates storage to the local variables. The number of local variables is supplied by the symbol table. The instructions "LOADVAL" and "LOADADR" set the value of and allocate storage to the parameters, and "CALL_FNP" sets the value of and allocates storage space to the stack linkage.

SLIM handles allocation of a stack segment differently. Allocation of a stack segment is shared by the code for a procedure call and the code for a procedure declaration. To be specific, the setting of and allocation of storage to the parameters and stack linkage is done before control of execution is transfered to the called procedure and storage for local variables is done before the execution of the first statement of the called procedure. Using as an example, the same procedure call as above, i.e., "pass(i,i)", the translation to SLIM of this call is

```
   PASCAL          SLIM            Comment

pass(i, i)      LIE2 PLE2       Pass the parameters
                UO PLE0         Pass the value of the static link
```

C@4 DX8002 Set up stack linkage and transfer control to the procedure

The SLIM instructions "LIE2 PLE2" set the value of and allocate storage to the parameters, "U0 PLE0 C@4 DX8002" set and allocate storage to the stack linkage. The local variable used by the procedure is allocated in the body of the procedure and is accomplished during variable declaration, i.e.,

PASCAL	SLIM	Comment
var: local1,		The unoptimized SLIM code
local2,		translation of this declaration
local3,		is M1 M1 M1 M1
local4: integer;	M4	

The difference lies in the way allocation of stack space to local variables is achieved. In PASCAL-S this is done by one instruction "MARKSTK", but in SLIM it is done for each declared variable. SLIM can have the same effect as the P-code, i.e., allocation is done by one instruction, only when the variable declaration does not contain array type variables.

Another difference between the two intermediate codes is in the translation of an access to an element of an array. Although, both intermediate codes use the same method, i.e., multiplicative subscript calculation, the translation is quite different.

PASCAL-S takes advantage of the accessibility of the symbol table at run-time. The translation of an array to P-code is simple and the work of accessing the bounds is left to the interpreter.

In contrast, SLIM does not have any access to the symbol table at

run-time. Instead, to make the bounds available at run-time, they are stored before the elements of the array in the local variable area of the stack segment allocated to the procedure where the array is declared. This way, access to the bounds of the array is achieved in the same manner as accessing a local variable.

To illustrate the difference, see the translation to SLIM and to P-code of an assignment statement involving arrays in the example program given at the end of this appendix.

Now, using the program below, we shall compare the two intermediate codes in terms of the number of instructions necessary to translate a PASCAL source code fragment. The program given does not solve any particular programming problem but is constructed so as to illustrate most of PASCAL constructs.

```
{ 1 }   program illustrate(output);
        type  day = (mon, tue, wed);
              r = record
                r1: integer;
                r2: array[0..2] of integer;
              end;
{ 2 }   var     count: integer;
                root: real;
            response: boolean;
                  ch: char;
               today: day;
                recd: r;
               table: array[0..1] of array[-1..0] of integer;
{ 3 }   function pass(valuep: integer; var varp: integer):integer;
{ 4 }   var local1, local2, local3, local4: integer;
        begin
{ 5 }    pass := valuep + varp;
```

```
          end;
          begin
{  6 }     count := 0;
{  7 }     root := 1.25;
{  8 }     ch := 'A';
{  9 }     today := mon;
{ 10 }     response := today = tue;
{ 11 }     recd.r1 := recd.r2[1];
{ 12 }     table[1,0] := table[0,-1];
{ 13 }     count := 11 * count + 20 mod count - 10;
{ 14 }     if (response) and (ch = 'E') or (root > 1.25) then count := 1
           else count := 0;
{ 15 }     case count of
            0: { nothing };
            1: { nothing };
           end;
{ 16 }     while count < 10 do
             count := count + 1;
{ 17 }     for count := 1 to 10 do { nothing };
{ 18 }     count := pass(count,count);
          end.
```

The translation of the program above to SLIM code and to P-code is shown below. The number of instructions (in parenthesis and bold letters) in the translation is given to illustrate the relative difference between the two intermediate codes, at least, in terms of the number of instructions needed to translate a PASCAL source code fragment.

PASCAL Source	SLIM Code	P-code
{ 1 }	@1: DO M1 (1)	(0)
{ 2 }	M6	
	L-6 PL2 PL0 PL3 P M3	
	L-9 PL1 PL0 PL2 PL0	
	PL-1 PL2 P M4 (16)	(0)

```
{ 3 }       @4: D2 M2 ( 1 )          ( 0 )

{ 4 }       M4 ( 1 )                 ( 0 )

{ 5 }                                LOADADR  2, 0
            L1E-6                    LOADVAL  2, 5
            PL1E-5 L!0               LOADIND  2, 6
            +H                       ADD_INT
            SE2 ( 5 )                STORE ( 5 )

{ 6 }                                LOADADR  1, 5
            L0                       LITERAL  0
            SE2 ( 2 )                STORE ( 3 )

{ 7 }                                LOADADR  1, 6
            L1.25E+00                IND_LIT  2
            SE3 ( 2 )                STORE ( 3 )

{ 8 }                                LOADADR  1, 8
            L'A                      LITERAL 65
            SE5 ( 2 )                STORE ( 3 )

{ 9 }                                LOADADR  1, 9
            L0                       LITERAL  0
            SE6 ( 2 )                STORE ( 3 )

{ 10 }                               LOADADR  1, 7
            L1E6                     LOADVAL  1, 9
            =1                       LITERAL  1
                                     EQ_INT
            SE4 ( 3 )                STORE ( 5 )

{ 11 }      LE7                      LOADADR  1,10
            PLE7                     LOADADR  1,10
            -1 SE1 P L!0 +H          REC_OFST 1
            PL0 +1 +H                LITERAL  1
            L!0                      STK_INDIR
            SH ( 12 )                STORE ( 6 )
```

```
{ 12 }          LE15                        LOADADR  1,14
                SE1 P L!0 +H
                PL0 +1                      LITERAL  1
                PLIE1 L!-6 *H               INDEX    2
                                            LITERAL  0
                PLH +H                      INDEX1   3
                PLE15                       LOADADR  1,14
                SE1 P L!0 +H
                PL0                         LITERAL  0
                PLH PLIE1 L!-6 *H           INDEX    2
                PL-1                        LITERAL  1
                                            NEGATE
                +H +H                       INDEX1   3
                L!0                         STK_INDIR
                SH ( 27 )                   STORE ( 13 )

{ 13 }                                      LOADADR  1, 5
                L11                         LITERAL 11
                *IE2                        LOADVAL  1, 5
                                            MULT_INT
                PL20                        LITERAL 20
                /*IE2                       LOADVAL  1, 5
                                            MOD_INT
                +H                          ADD_INT
                -10                         LITERAL 10
                                            SUB_INT
                SE2 ( 7 )                   STORE ( 11 )

{ 14 }          LIE4                        LOADVAL  1, 7
                F@6 PLIE5                   LOADVAL  1, 8
                ='E                         LITERAL 69
                                            EQ_INT
                /\H                         AND_BOOL
                T@6 PLIE3                   LOADVAL  1, 6
                *>1.25e+00                  IND_LIT  2
                                            GT_REAL
                \/H @6:                     OR_BOOL
                F@7                         COND_JMP 68
                                            LOADADR  1, 5
```

```
                L1                          LITERAL  1
                SE2                         STORE
                J@8                         JUMP    71
                @7:
                                            LOADADR  1, 5
                L0                          LITERAL  0
                SE2                         STORE ( 17 )
                @8: ( 15 )

{ 15 }          LIE2 P J@9                  LOADVAL  1, 5
                                            SWITCH  75
                @11: J@10                   JUMP    80
                @12: J@10                   JUMP    80
                @13: L1 CIG87 D1 Q
                @9: L3 ?S
                        D@13
                    D0  D@11                CASELAB  0
                                            CASELAB 73
                    D1  D@12                CASELAB  1
                                            CASELAB 74
                                            JUMP     0 ( 9 )
                @10: ( 11 )

{ 16 }          @14:
                LIE2                        LOADVAL  1, 5
                <10                         LITERAL 10
                                            LT_INT
                F@15                        COND_JMP 90
                                            LOADADR  1, 5
                LIE2                        LOADVAL  1, 5
                +1                          LITERAL  1
                                            ADD_INT
                SE2                         STORE
                J@14                        JUMP    80 ( 10 )
                @15: ( 8 )

{ 17 }                                      LOADADR  1, 5
                L1 SE2                      LITERAL  1
                L10                         LITERAL 10
```

```
                    J@16                            FOR1UP  99
                    @17: LIE2                       LOADVAL 1, 5
                    PL6                             LITERAL 6
                    CIG71 D2                        WRITE_2 2
                    CIG67 D0                        WRITELN
                    LIE2 +1 SE2 LH                  FOR2UP  94 ( 9 )
                    @16: P >=IE2 T@17 M-1 ( 16 )

{ 18 }                                              LOADADR 1, 5
                                                    MARKSTK 44
                    LIE2                            LOADVAL 1, 5
                    PLE2                            LOADADR 1, 5
                    U0 PLE0 C@4 DX8002              CALL_FNP 6
                    SE2 ( 6 )                       STORE ( 6 )
```

The given program has an equivalent of 106 P-code instructions and an equivalent of 136 instructions in SLIM. The D (pseudo)instruction in SLIM in not included in the total number of SLIM instructions because the equivalent of D (pseudo)instruction in the assembly language of the target machine is actually an assembler directive and not an assembler instruction.

We can observe that the SLIM code equivalent requires more instructions to access an element of an array. This is because, in P-code the computation of the address of the element of an array to be accessed is done by the interpreter (note that the interpreter has access to the symbol table where the bounds are stored).

When it comes to the translation of an assignment statement where the destination is an entire variable, the SLIM code equivalent is one instruction less than the P-code equivalent.

In the translation of expressions, there are some constituents of an expression that require only one instruction in SLIM but require two in P-code. For example, the SLIM code "+1" where an equivalent P-code instructions are "LITERAL 1" and "ADD_INT".